Order Number 9020794

# Expert systems and test plan generation

Papavasiliou, Ioannis, Ph.D.

City University of New York, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

A

# EXPERT SYSTEMS AND TEST PLAN GENERATION

by

IOANNIS PAPAVASILIOU

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York.

1990

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

| | |
|---|---|
| 1/10/90 | |
| Date | Chair of Examining Committee |
| 1/10/90 | |
| Date | Executive Officer |

Professor Michael Anshel

Professor Howard Wasserman

Professor T. C. Wesselkamper

Supervisory Committee

The City University of New York

# A b s t r a c t

**EXPERT SYSTEMS AND TEST PLAN GENERATION**

**by**

**IOANNIS PAPAVASILIOU**

**Adviser : Professor Howard Rubin**

The purpose of this work is the design of an expert system that supports the automation of the contents of the test plan, improving in this way the productivity, quality and reliability of software, which subsequently reduce the software cost. Chapter I gives some background of the problems that currently exist in the development of software systems and drive the software costs up. To overcome these problems, we propose a solution: to automate the labor intensive functions in the software development process using expert systems, and restrict our work in constructing the test plan using this approach. In Chapter II the scope of the system is defined. It accepts as input the flow graph of a program written in a simple subset of PASCAL and produces as output the number and definitions of test cases, the test

coverage, the number of predicted defects and the estimated reliability. In Chapter III we discuss several expert systems that are currently supporting software development, and we also summarize the related work on software testing.

Chapter IV is the core of this work. The design of the system is described step by step. The system uses the data selection criterion "all-uses", from the family of criteria proposed by Rapps and Weyuker, to select the paths to be tested and then forms the path constraints by using symbolic evaluation. The implementation of the first part, i.e., the selection of the test paths, and some actual results are given in the appendixes. Chapter V is a brief discussion on the possible future research directions for this project.

Dedicated to my parents,
YORGOS and EVANGELIA PAPAVASILIOU

## Acknowledgments

I would like to thank all these people who helped me to overcome the difficulties throughout this work, and to make it possible. Professor Howard Rubin, my advisor, for his invaluable technical guidance and moral support. Professor Thomas C. Wesselkamper for his suggestions on various problems, and for his time and patience to correct all the grammatical mistakes in the dissertation. Professor Michael Anshel for his guidance and suggestions, and for his moral support when problems arose. Professor Howard Wasserman for allowing me to audit his class "Logic Programming" at Queens College, and for all the information and material on Prolog and its implementation. I would also like to thank my parents, Yorgos and Evangelia Papavasiliou, who provided unwavering support and encouragement throughout my education.

## Table of Contents


Chapter I

Introduction
- Background 1
- Problem 4

Chapter II

Outline and limitations of the system
- Scope 5

Chapter III

Summary of Current Knowledge
- Related Work on Expert Systems 9
- Related Work on Testing 12
  - Test Case Generation 12
  - Software Reliability 24
  - Defect Prediction 28

Chapter IV

Design of the system
- Overview 30
- Design 33
  - Test Case Generation 33
  - Software Reliability 42
  - Defect Prediction 44

Chapter V

Conclusion
- Summary 46
- Future Work 47

Appendix A
- The program shell 49
- Documentation 65

Appendix B
Examples 85
Explanation of the examples 88

Chapter VI

Bibliography
Cited References 129
Uncited References 134

## Table of Figures


Figure 1 : Software demand, productivity growth 2
Figure 2 : Software - hardware costs 2
Figure 3 : Test data generation flow - general diagram 31
Figure 4 : Test data generation flow - parts supported by the system 32
Figure 5 : Software reliability estimation diagram 32
Figure 6 : P - Use Example 85
Figure 7 : C - Use Example 86
Figure 8 : Mixed - Case Example 87

# CHAPTER I

# INTRODUCTION

## 1.1 Background

The development of software systems is usually straightforward, but often results in the project which is over budget, both in terms of cost and time, or even results in flawed products. As we move towards complex software applications, programmers fall farther behind the demand and their results are of poorer quality. High demand and comparatively low productivity drive software costs up. The need for software is growing exponentially, but productivity is only rising at a rate of about 5% a year [12]. Figure 1 shows the growth per year of the software demand, software productivity, and software personel, from 1980 to 1990. Although progress is continuously being made in software technology, it appears slow, compared to the developments in hardware technology.

Thus, there is a necessity for a solution that makes software costs drop as rapidly as hardware costs do. Figure 2 reflects the relationship between the hardware and software costs. There is no single development, in either technology or in management, that promises improvement in productivity,

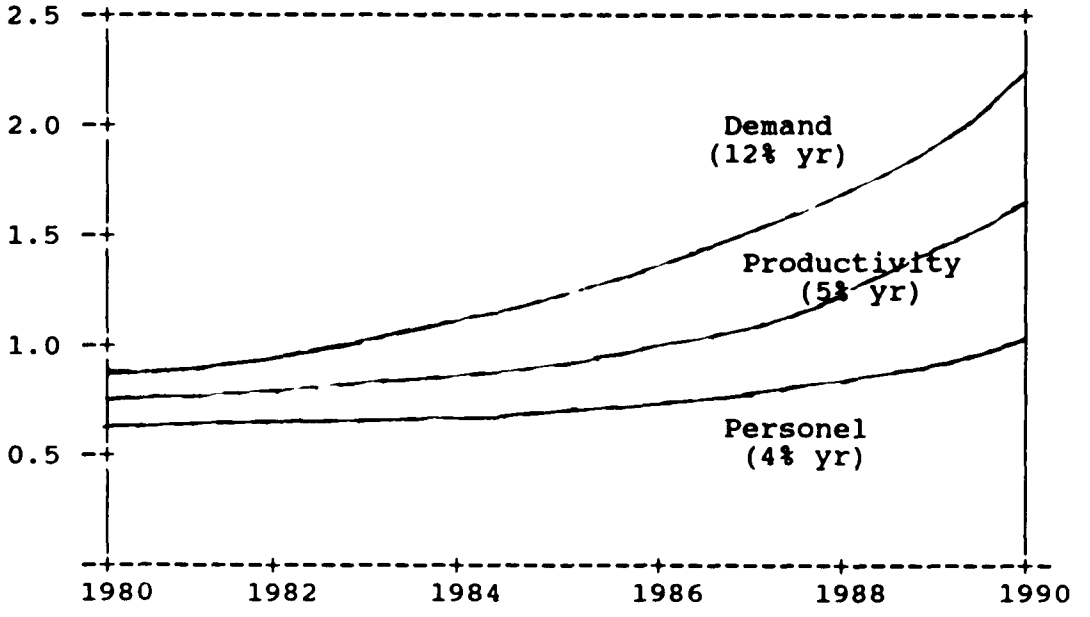



**Fig. 1**

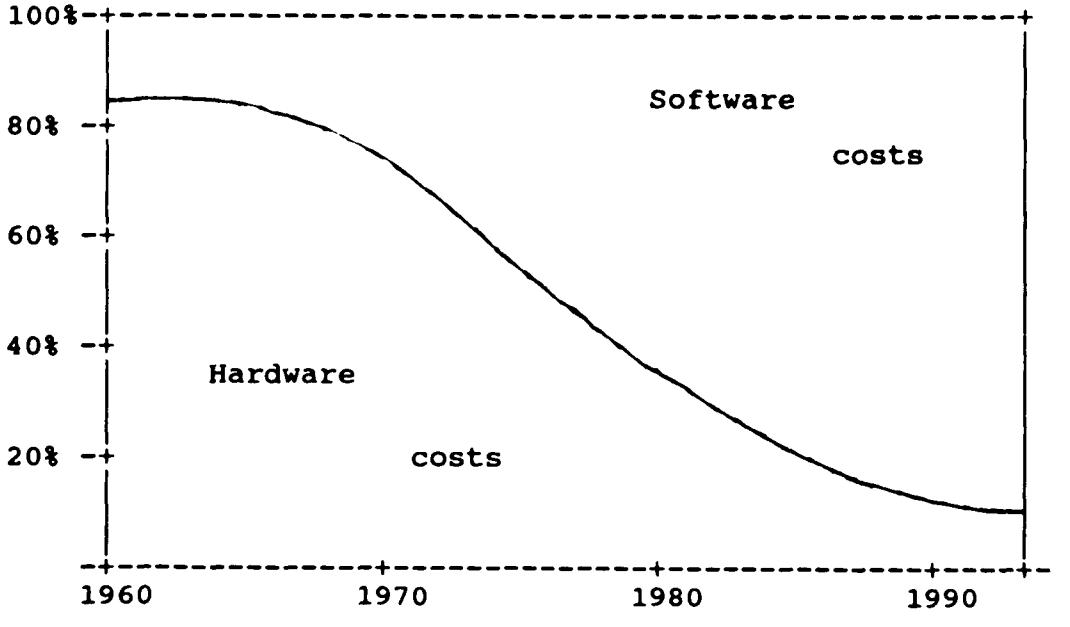



**Fig. 2**
SOURCE : Electronic Design, 1981.

reliability, or simplicity [2]. Therefore, to overcome these problems, we must apply several technical developments that help us manage the complexity of software solutions.

One of these developments, the technology for building expert systems, can be applied to the software engineering task in many ways [2]. The recent history of expert systems shows that tasks comparable in technology and complexity to the tasks one performs in building software can be automated. [33]. Any function in the software development life cycle that needs labor-intensive effort is a good candidate for automation, i.e., it could be transferred from a manual operation to an automated process. Since a totally automated software development process is not currently available, the automation of some of its functions would improve productivity, quality, and reliability with a subsequent gain being reduced cost.

The test plan, which guides the testing activities efficiently, considered as a part of the planning activity is primarily a manual operation. Therefore it must be automated to yield an improvement in the software development process. Until now there is no expert approach for constructing the test plan, so producing software is more costly.

## 1.2 Problem

The problem addressed in this dissertation is the automation of the process which generates the test plan. This function should take place early in the system development life cycle as part of the planning activity.

# CHAPTER II

# OUTLINE AND LIMITATIONS OF THE SYSTEM

## 2.1 Scope

This study is concerned with the automated generation of the test plan, which is an integral part of the overall project development planning. The system is restricted to the structural testing of an individual module, i.e., the control structure of the module is used as the basic source of information for generating the test cases. Module or unit testing is the validation of the module in an isolated environment. Each module implements a single independent function and has a single entry point and a single exit point. The size of a module is an important factor, since it is related to the ease of testing, e.g., the number of paths. To simplify the problem, we assume that modularization has been applied, i.e., the major subfunctions to be accomplished have been identified and the module size is small enough. Since there is a relationship between the number of errors and the size of the module [13], size is considered in the dissertation. The module's function is the transformation from input to output and its logic is the description of control flow within it, represented by a flow graph. Each node

represents a program segment, i.e., sequential code with a single entry and a single exit, and each edge indicates flow of control between two segments. The node containing the first statement of the module has no predecessors and is called the source, while any node with no successors is a sink node. A test case in the program corresponds to a path from the source to a sink which represents the execution sequence evoked by the test case.

The system's focus is on the contents of the test plan, namely, the number of test cases, the definition of test cases (the input values, the expected results, and the requirement(s) to be verified), defect prediction, potential test coverage, and the estimated reliability. There are only simple variables and the types of statements allowed are assignment statements, conditional and unconditional transfer statements, and input and output statements. Also there are paths from the source to every node in the flow graph, and on every path of the form ( $n_1,\ldots,n_k,n_1$ ), i.e., on every loop, there is some segment that contains a conditional transfer to a segment that is not on the path. We assume that coincidental correctness does not occur for any test point, i.e., if a test point follows an incorrect path, the output values are not the same as if that test point were to follow the correct path. The path condition is the compound condition

that must be satisfied by the input data point in order for the control path to be executed. It is the conjunction of the individual predicate conditions generated at each branch point along the control path. If the path condition is not satisfied by any input value, the control path is infeasible, and is of no use in testing the module. The path condition is a set of equality and inequality constraints on the input variables. To generate test data that satisfy the path condition requires that the system be capable of solving arbitrary sets of inequalities. However, determining such a solution is an undecidable problem [9]. Thus, the path feasibility problem is also undecidable. Due to the various forms of constraints and type requirements on inputs, an integrated collection of inequality solution techniques is usually needed [28]. A usual hypothesis is that the inequalities used are relatively simple and often linear, which is a limitation for the testing strategy. White and Cohen [40], based on research works of several people, argue that nonlinear predicates are rarely encountered in data processing applications. Also, nonlinear predicates, except the simplest, pose problems of extra processing which maybe preclude testing. In this case of simple and linear inequalities, linear programming techniques can be used to solve a system of linear constraints. If the above hypothesis is proved false,

other methods can be applied to solve the set of constraints, such as the conjugate gradient algorithm. The system is able to deal with linear and nonlinear constraints, and with constraints that contain alphanumeric variables and literals. Since no method can solve all arbitrary systems of constraints, any chosen algorithm sometimes fails, and so in some cases consistency or inconsistency cannot be determined [6]. Missing paths, i.e., the module does not contain the corresponding path of a special case, represent another fundamental limitation to a testing strategy based on the structure of the program. It is assumed that such an error is not associated with the path being tested, since in order to find it, we have to look at supplementary sources such as software specification. No attempt is made to deal with any of the administrative details.

# CHAPTER III

# SUMMARY OF CURRENT KNOWLEDGE

## 3.1 Related Work on Expert Systems

An expert system is an intelligent computer program which consists of two parts : a knowledge base containing the facts and heuristics about a particular discipline and a collection of inference procedures applying reason to use the knowledge base. An integrated software development expert system automates the software development process from the requirements phase through the verification and validation phase.

Several such systems have been designed and are currently supporting software development. They are categorized in [20] into two major areas: systems that concentrate on one part of the software problem such as coding, and systems that concentrate on the entire software development process for specific application such as symbolic computation. From that source [20] we give the members of each category and their functions. In the first class we have:

(a) Programmer's Apprentice - keeps track of details, as the programmer designs the program, and assists in documentation, verification, debugging, and modification.

(b) PHENARETE - takes a LISP program as input, detects certain errors and suggests corrections.
(c) SAFE - takes an informal specification as input and attempts to produce an unambiguous specification in a formal language.
(d) PROTOSYSTEM I - accepts a specification input in a high level language and attempts to generate an efficient implementation in PL/1.
(e) DEDALUS - converts a high level complete specification to an implementation in a simple LISP-like language.
The members of the second class are:
(a) PSI - automates the development from a functional input description through implementation in LISP.
(b) NLPQ - accepts inputs in English language and produces simple queueing in GPSS.

Finally, the system PLANMACS has been developed [30] for the automation of plan generation, which is among the most manually intensive areas in project management. It consists of three major components : PLANNER, SCHEDULER, and TRACKER. The PLANNER takes as input a macro estimate of project effort and, using descriptive information, constructs the project plan, which the user can modify by adding or deleting activities. The SCHEDULER places the plan onto an actual calendar and real staff resources are assigned to the

project. The TRACKER produces the tracking reports by which the progress of the project can be judged, and detects significant variances. The PLANMACS system with the combination of continuous estimation, "expert" planning, intelligent scheduling with tracking, provides an integrated support environment for project management, further improving productivity.

The test planning is an integral part of the overall development planning, and starts during the initiation of the project. The emphasis in test planning should be on defect removal. This implies the efficient utilization of a variety of test techniques and the consideration of many factors such as size (measured in thousands of lines of code), and cyclomatic complexity (defined as the number of decisions plus one), so that the plan is both feasible and effective. The process of test plan generation is ongoing. As the process progresses new test activities are added and new test cases are generated.

## 3.2 Related Work on Testing

### 3.2.1 Test Case Generation

The test plan describes the complete software test program and is based on the Software Requirements Specification. The ultimate objective of test planning is to maximize the number of errors found, with respect to the amount of testing. Modules contain two types of errors: computation errors and domain errors. A domain error occurs when a specific input follows the wrong path due to an error in the control flow of the module. A computation error is contained in a path when a specific input follows the correct path, but an error in some assignment statement causes the wrong function to be computed for one or more of the output variables [17]. In practice, such classification may not be simple. First, multiple errors of various types may occur. Second, an assignment error may cause both domain changes and computation errors. As mentioned in [39], Chandrasekaran argues that the classification of an error may depend upon one's perception. For instance, consider the use of "<" instead of ">" in a predicate. This could be considered as a domain error or as a computation error. In the first case, the predicate needs to be modified, while in the second, the functions computed on

the two branches need modification. Which of these modifications is simpler is something that is not directly captured in the definitions of the error types. In spite of the above, this classification remains a viable and useful approach [39]. Domain errors can be further divided [5] into path selection and missing path errors. The first type occurs when a module incorrectly determines the condition under which the path is selected, and the second type occurs when the module does not contain the path corresponding to a special case.

The goal of module testing is to detect errors. Testing the module on all possible input values would provide the most complete picture of its behavior, which should increase our confidence that the module behaves according to its specification. But the input domain is usually very large and makes exhaustive testing infeasible. The usual procedure is to select a relatively small subset which is representative of the entire input domain [29]. Any finite subset of the module's input domain can be considered as a set of test data. A test selection criterion specifies conditions which must be fulfilled by a set of test data. Any set of inputs satisfying these conditions is said to be a test selected by this specific criterion [38].

Ideally, our goal is to construct tests that uncover

all errors in a program. Then we can conclude that, since the program produces correct results for the test data, it produces correct results for any data in the input domain. An ideal test, as defined in [15], consists of a set of test data, such that there is an input in the program's domain for which an incorrect output is produced if and only if there is some element of the test set on which the program is incorrect. An ideal criterion, i.e., one which always selects ideal tests, has the following two properties:

(a) it is valid, i.e., it does not exclude the selection of test data capable of revealing some error. In other words, whenever the program is incorrect, the criterion selects at least one test set which is not successful for the program. By successful test we mean that the program is correct for every element of the test set;

(b) it is reliable, i.e., either every test selected by the criterion is successful, or no test selected is successful. Reliability refers to the consistency with which results are produced, i.e., the criterion must insure selection of tests that are consistent in their ability to reveal errors [15].

The successful execution of an ideal test is equivalent to a proof of correctness. Although test sets satisfying such conditions can be generated for certain classes of errors and programs [40], discovering ideal sets of test data is

in general an impossible task. This is due to the fact that it is difficult, and impossible in general, to find proofs of program correctness [38]. Howden [17], has proven the test selection problem to be undecidable. Although we know that there exists a finite test set which reliably determines the correctness of a given program over its entire input domain, no algorithmic method exists for constructing such a set for an arbitrary program. In the construction of such a test several other problems are encountered [38]. A criterion is guaranteed to be both reliable and valid if and only if it selects the entire domain as one test, but exhaustive testing is clearly impractical. In addition, a criterion which is reliable or valid for a program is not necessarily so for a slightly different one. Finally, there is a dependence between validity and reliability. At least one of the two properties must hold for any criterion, because an invalid test criterion exposes no errors, i.e., all of its tests are successful. Weyuker and Ostrand [38], in an attempt to overcome these difficulties, introduced the notion of a revealing domain that allows the division of the domain into smaller, more manageable pieces and relativizes to errors the notion of revealing. This leads to the revision of the goal from showing program correctness to showing the absence of specified errors, something easier

easier to attain.

Goodenough and Gerhart [15], demonstrated the unreliability of test criteria based solely on the internal structure of a program. They suggested the following sources of information for deriving test data:

(a) the general requirement a program is to satisfy;

(b) the specification of the program;

(c) general characteristics of the implementation method, including special conditions relevant to data structures and how data are represented; and the specific internal structure of an actual implementation.

The process of test data selection is divided into two phases, creation of test data to test the function and creation of test data to test the details of design and implementation. Functional based testing concentrates on the module's specifications and test data is derived to test the program as a black box, i.e., to test if the module meets all its functional requirements. This may consist of:

(a) Representative elements from each valid input class. The classes are obtained by dividing the domain of all valid inputs, so that elements from the same class are processed similarly by the module. The relational and type constraints on input variables define the allowable input domain.

(b) Data at the boundaries of each valid input class.

(c) Data which require special processing.

(d) Representative elements to generate outputs within and at the boundaries of each output class. The output domain is defined similarly to the input domain, but for the output variables.

(e) Invalid data, which is found by the above analysis techniques.

Black box testing is not effective for detecting certain types of errors, e.g., errors in the handling of special conditions.

Design/implementation based test data is derived to test the program as a white box, and the control structure is used as the basic source of information. This test data covers all the aspects of the module not covered previously and may consist of:

(a) test data elements that can test the data structures, algorithms, design decisions, and internal functions of the module;

(b) test data that can test the algorithm with standard and special values of data structures;

(c) test data that can stress the algorithm with extremal values of data structures [19].

The main shortcoming of structural testing is that tests are generated using the possibly incorrect code, so certain types

of errors, especially errors in the specifications, are hard to detect.

In the case of white box testing, in order to determine whether the coverage is sufficient, it is necessary to have a structural coverage metric. We select a finite set of module paths, using the control flow graph, satisfying one or more coverage criteria, and then find the input data which cause each of the chosen paths to be selected. A criterion provides a measure of the number of structural units which are fully exercised by the test data [1]. The most usual are [27]:

(a) statement coverage, i.e., execution of all statements in the graph;

(b) branch coverage, i.e., encounter all exit branches of each decision node in the graph;

(c) multiple condition coverage, i.e., achieve all possible combinations of simple boolean conditions at each decision node in the graph;

(d) path coverage, i.e., traverse all paths of the graph.

The above criteria are related as follows: (b) implies (a), and is a minimal standard in structural testing. Also, (c) implies (b) and (a), and does not require following any more paths than (b) does. But (c) is more difficult, because it requires more data to achieve all possible combinations.

On the other hand, it is much easier than (d), which is difficult, if not impossible, to achieve, because the number of paths may be infinite [24].

A class of test data selection criteria has been proposed by Rapps and Weyuker [29], and is based on data flow analysis, which examines the branch and loop structure of a program. Similar criteria were proposed in [21] and [25]. In this case, the number of selected paths is always finite and is chosen in a systematic manner. First, we give some preliminary definitions from [29], which are needed to define this family of data flow testing criteria. Each variable occurrence is classified as being a definition or a use. A definition is the occurrence of a variable, in the left side of an assignment statement or in an input statement. A use is the occurrence of a variable, in the right side of an assignment statement or in an output statement, (c-use), or in the predicate portion of a conditional transfer statement, (p-use). A c-use is a global c-use if there is no preceding definition of the variable within the segment in which it occurs. Otherwise it is a local c-use. Since the value of the variable occurring in the predicate portion of the conditional transfer statement determines which branch is to be followed, p-uses are associated with edges rather than with the segment in which the predicate portion occurs. Thus,

there is no need to distinguish between local and global p-uses. A simple path is one in which all nodes, except possibly the first and the last, are distinct. A loop-free path is one in which all nodes are distinct. A path $(i,n_1,\ldots,n_m,j)$ $m>=0$, containing no definitions of variable x in nodes $n_1,\ldots,n_m$ is called a def-clear path with respect to x from node i to node j and from node i to edge $(n_m,j)$. A definition of a variable x in node i is a global definition, if it is the last definition of x occurring in 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 path $(n_1, \ldots,n_j,n_k)$ is a du-path with respect to a variable x if $n_1$ has a global definition of x and either:

(1) $n_k$ has a c-use of x and $(n_1,\ldots,n_j,n_k)$ is a def-clear simple path with respect to x, or

(2) $(n_j,n_k)$ has a p-use of x and $(n_1,\ldots,n_j)$ is a def-clear loop-free path with respect to x.

Now we obtain the family of data flow testing criteria, using P as a set of paths through the flow graph:

(a) P satisfies the all-nodes criterion if every node of the graph is included in P.

(b) P satisfies the all-edges criterion if every edge of the graph is included in P.

(c) P satisfies the all-definitions criterion if every global definition is used.

(d) P satisfies the all-p-uses criterion if a path from every global definition to each of its potential p-uses is included in P.

(e) P satisfies the all-c-uses/some-p-uses criterion if for every node i and every x which has a global definition in i, P includes some definition clear path with respect to x from i to every node j, such that x has a global c-use in node j, and for which there is a definition clear path with respect to x from i to j. If there are no such nodes, then P must include a definition clear path with respect to x from i to some edge (j,k), which contains a p-use of x, and for which there is a definition clear path with respect to x from i to j.

(f) P satisfies the all-p-uses/some-c-uses criterion if for every node i and every x which has a global definition in i, P includes a definition clear path with respect to x from i to every edge (j,k), such that (j,k) contains a p-use of x, and for which there is a definition clear path with respect to x from i to j. If there are no such edges, then P must include a definition clear path with respect to x from i to some node j, such that x has a global c-use in node j, and for which there is a definition clear path with respect to

x from i to j.

(g) P satisfies the all-uses criterion if a path from every global definition to each of its uses is included in P.

(h) P satisfies the all-du-paths criterion if for every node and every variable x, which has a global definition in that node, every du-path with respect to x is included in P. If there are multiple du-paths from a global definition to a given use they must all be included in paths of P.

(i) P satisfies the all-paths criterion if every path from the source node to a sink node is included in P.

In the above hierarchy the all-nodes criterion corresponds to statement coverage, and the all-edges criterion "roughly" corresponds to branch coverage. The characterization of the second correspondence ("roughly") is due to the way the predicates, controlling a branch, are placed in the graph. Branch expressions are placed on edges instead of in separate branch nodes [37]. Schematically, the hierarchy is as follow: (i) => (h) => (g) => [ (e),(f) ] => (c), and (f) => (d) => (b) => (a). The decision of which criterion to use as a basis for test data selection depends on several factors, such as the size of the program, time and cost requirements, etc. They can be used to bridge the gap between the requirement that every branch be traversed and the frequently impossible requirement that every path be traversed [29].

Another approach to testing, called mutation testing, is proposed in [10] and uses some techniques from mutation analysis. Mutation testing is a member of the set of error driven strategies, which use known errors to guide the generation of test cases. The primary objective of mutation analysis is to evaluate the degree to which a test set exercises a program, rather than the initial generation of the test. Given are a program P and a set of test cases T, whose adequacy is to be determined. The program is executed on test data and is assumed to give correct answers, because if it does not then certainly it is in error. Assuming no errors are exposed, the program may be in error but the data is not sensitive enough to distinguish the error. A number of alternative programs, called mutants of P, are produced by small changes in P, and T is executed on each mutant. Continuing this analogy, if at any point, P and this mutant produce different output, the mutant has died. If identical responses are obtained the mutant survives. If a large number of the mutants survive, then the set T is insufficient and additional test cases are needed. If the number is small, and the set of incremental operators is sufficient, then the test set must have been tied closely to the form and function of the program. The surviving mutants provide a new method of generating test data to eliminate these mutants.

This leads to an iterative testing process, terminating when nearly all mutants have been eliminated. Mutation testing refers to the construction of tests designed to distinguish between mutant programs that differ by the occurrence of simple errors (for instance, "A.EQ.B" instead of "A.LE.B"). But, since it is unrealistic to assume that we can enumerate all possible errors a programmer could make, an alternative idea is considered in [3]. The construction of mutant programs is a sequence of applications of mutant operators. Each operator is a one-to-many mapping which takes as input a program and produces a sequence of alternative programs, (in practice, descriptions of the differences are produced). For example, if we consider the mutant operator "Arithmetic operator replacement", then in every place an arithmetic operator is used, it can be replaced by all other operators of the same type (arithmetic), and a syntactically correct program results. Disadvantages of mutation analysis include the potentially large number of runs needed and questions regarding the validity of its assumptions.

### 3.2.2 Software Reliability

The general goal of testing is to affirm the quality of the software. Software reliability, as defined in the IEEE

Standard Glossary of Software Engineering Terminology, 1983, is the probability of failure-free operation of a software component or system in a specific environment for a specified time. A failure is any departure of program output from requirements as the program is executed. Reliability measurement is divided into three categories [4]: prediction, estimation, and assessment. Prediction covers the life cycle phases from software requirements to coding/unit testing, estimation covers the test and evaluation phases, and assessment takes place during operation and maintenance phases. Software reliability is a useful measure in planning and controlling the testing resources, which can be done by balancing the additional cost of testing and the corresponding improvement in software reliability.

There are some factors, that must be considered in estimating reliability [4]. The software test process must be examined to determine which test methodology or testing techniques are used and how effectively they are employed. It becomes necessary to consider the subdivision of testing - module, integration, system, and installation testing - because testing may not be uniform from phase to phase and thus the probability of uncovering a software failure can be dependent on the test phase. While it is impractical to achieve 100% test coverage, some indication is needed to

determine how extensively testing has been performed.

A commonly used approach for measuring software reliability is via an analytical model whose parameters are generally estimated from available data on software failures. Reliability and other relevant measures are then computed from the fitted model. A number of analytical models have been proposed to address the problem of software reliability measurement. These approaches are classified in [14] according to the nature of the failure process, as indicated below.

1) Times between failures models :

The interest of this class of models is in the times between failures. The most common approach is to assume that the random variable $T_i$ denoting the time between $i^{th}$ and $(i-1)^{st}$ failures follows a known distribution whose parameters depend on the number of faults remaining in the program during this interval. Note that the distribution of a random variable Z is the function (defined over the value set of the variable) which associates with each possible value z of Z the probability P(Z=z) that the value z will occur. Estimates of the parameters are derived from the observed values of times between failures and estimates of software reliability are then obtained from the fitted model.

2) Failure count models :

The interest of this class of models is in the number of

faults or failures in specified time intervals. The failure counts are assumed to follow a known stochastic process with a time dependent discrete or continuous failure rate. A stochastic process is an arbitrary infinite family of real random variables ( $z_t$, t in T ), where t is the time parameter, T is the time interval involved, and $z_t$ is the observation at time t. Parameters of the failure rate can be estimated from the observed values of failure counts or from failure times. Then estimates of software reliability can be obtained from the relevant equations.

3) Fault seeding models :

The basic approach is to "seed" a known number of faults in a program which is assumed to have an unknown number of faults. The program is tested and the observed number of seeded and prior existing faults is counted. Then an estimate of the faults existing in the program prior to seeding is obtained and used in the measurement of software reliability.

4) Input domain models :

The input domain is partitioned into a set of equivalence classes, each of which is usually associated with a program path. An estimate of program reliability is obtained from the failures observed during physical or symbolic execution of the test cases sampled from the input domain.

The typical environment during unit testing phase is

such that the test cases do not form a representative sample of the operational usage distribution. The input domain based models are applicable during this phase, except that matching the test profile to operational usage distribution could be difficult. Also fault seeding models are applicable, provided it can be assumed that the existing and seeded faults have equal probabilities of being detected. The time dependent models do not seem to be applicable, because the independent times between failures assumption is seriously violated [14].

### 3.2.3 Defect Prediction

Several methods have been proposed in the literature to estimate the number of errors in a program. Lipow [22], estimates the number of errors per line of code, based upon Halstead's software science relationships. It is shown first that the number of errors in a program increases with the number of lines of code in the program, and consequently that the number of errors in a program is a linear function of terms in $P \ln P$ and $P \ln^2 P$, where $P$ is the number of lines of executable code. The result is a form readily usable, once the language is decided upon and the number of lines of code is estimated. A similar approximation, based on [22], was developed in [34]. It is easier, more accurate,

and simpler to use than the one in [22]. Also, Schneider in [32], presents some estimators which are extensions of the Halstead software science estimators in that they use software metrics easily obtainable. In particular a formula is given that can be used to predict the number of errors in a module once the amount of code to be developed is estimated. Gaffney [13] has shown that a relationship exists between the number of errors and the number of lines of code, regardless of the language used. Further, it was shown in [13] that estimates, based on the number of lines of code that comprise a module, are as good or better than ones based on additional information, and can be done at the time an estimate of the amount of code to be developed is made. This is important because these estimates are of interest early in a software development project and information such as vocabulary size is not available.

# CHAPTER IV

# DESIGN OF THE EXPERT SYSTEM

## 4.1 Overview

An expert system has been designed, that supports the automation of the contents of the test plan. Figure 3 shows the complete flow for the test data generation process [18], and figures 4,5 the parts supported by the system. The system consists of three components: a knowledge base, an inference mechanism, and a user interface. The knowledge base comprises the knowledge that is specific to the domain of testing in general and to the application on hand, including simple facts, rules that describe relationships, and methods and heuristics for solving problems in the domain of testing. The flow graph of the module is used as the main source from which knowledge about the particular application is acquired. In terms of structural testing, the branch testing is considered to be a minimal requirement while the path testing is considered to be impractical. A test data selection criterion is chosen such that it subsumes branch testing and stays short of path testing. The chosen test data selection criterion is incorporated into the knowledge base. The inference engine actively uses the knowledge in the base and in-

**General Diagram**

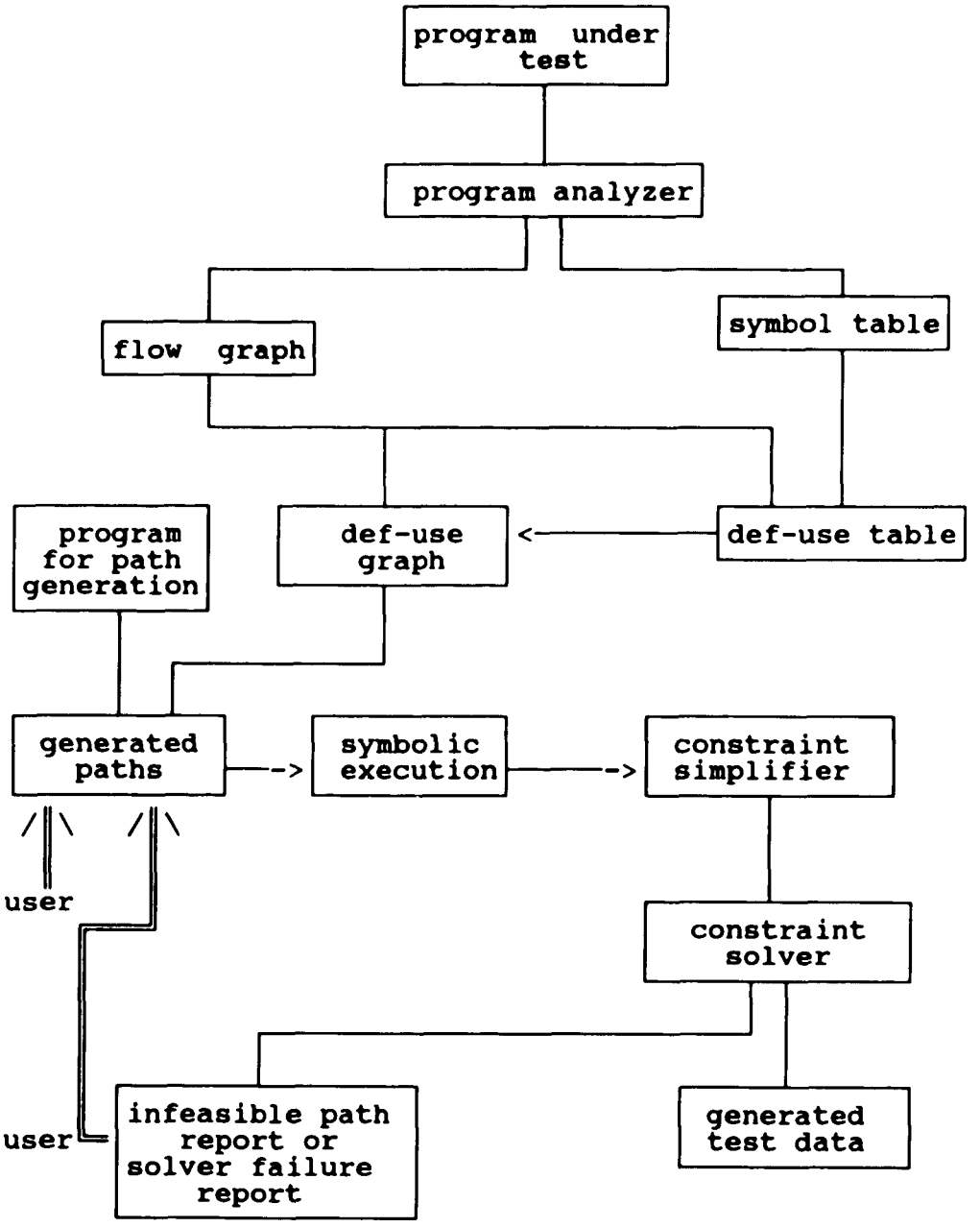



Fig. 3

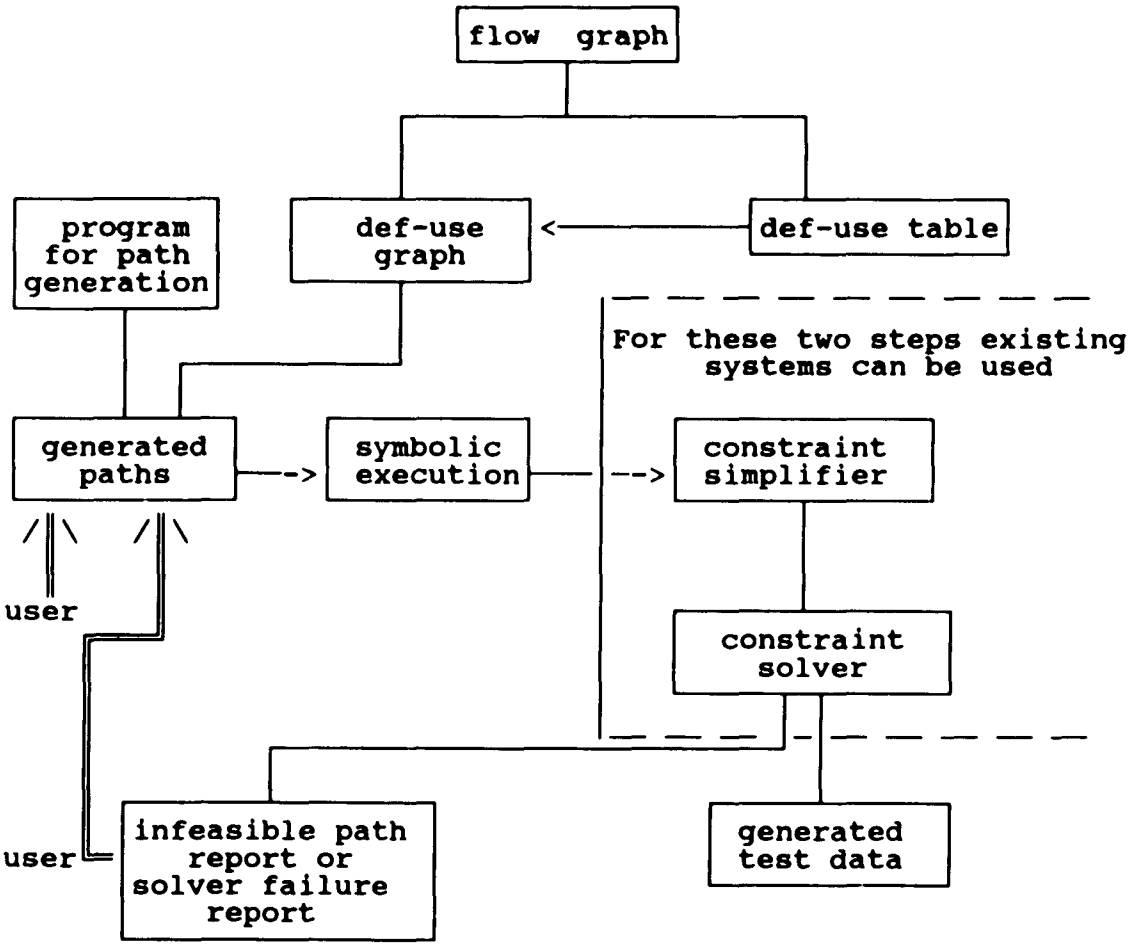

flow graph
program for path generation
def-use graph
def-use table
For these two steps existing systems can be used
generated paths
symbolic execution
constraint simplifier
user
constraint solver
user
infeasible path report or solver failure report
generated test data

Fig. 4

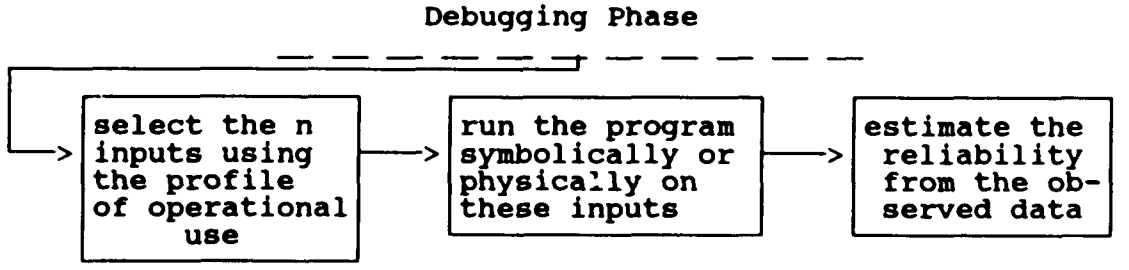

Debugging Phase
select the n inputs using the profile of operational use
run the program symbolically or physically on these inputs
estimate the reliability from the observed data

Fig. 5

formation provided by the user to infer new knowledge. The user interface is used for communication between the user and the system. Through the user interface, the user can enter facts about a specific situation relevant to the system's subject domain and can ask the system questions within its subject area. It also provides the user with an insight into the problem solving process carried out by the inference mechanism. If-Then rules, also called production rules, are used to represent the knowledge.

## 4.2 Design

### 4.2.1 Test Case Generation

The test data flow criteria can be used to bridge the gap between the requirement that every branch be traversed and the frequently impossible requirement that every path be traversed [29]. The test data selection criterion all-uses, in the family of criteria proposed by Rapps and Weyuker [29], is used in our system. It asks that all interactions, between a use ( either c-use or p-use ) and a definition that reaches it, be tested. It is the central criterion of the family, and is similar to strategy I [21], and required element testing [25]. The all-uses strategy is redefined, by associ-

ating all the p-uses for a predicate with the segment in which the predicate occurs, to include branch testing [37]. It may require $O(n^2)$ test paths, where n is the number of segments, and offers needed improvement over branch testing while its cost remains reasonable [26]. Loops are exercised until the def-use chains are exhausted, so there is no limit on loop iterations and the actual number of iterations is finite, because the number of chains is finite [21]. For some programs all-uses may not test all possible combinations of predicate outcomes [29].

The first step of the test data generation process is to derive the set of paths that must be tested according to the given strategy. Given a subject program, its flow graph is obtained. There are several programs that can be used to perform such an analysis on FORTRAN programs or programs written in a subset of PASCAL [6], [11], [28], [36]. Since the language accepted by our system is a subset of PASCAL, we use the ASSET program for that purpose. First a label table is constructed, which associates label occurrences and their uses. Second, the statements are classified as labeled, unlabeled, conditional, unconditional, or other. This information, along with the label table is used to divide the program into segments and to produce the flow graph [11]. The flow graph is the input to our system and provides the neces-

sary information, specific to the problem. This information is represented by facts, and identifies the possible flow of control, node id, and the statements contained in each node. There are production rules in the knowledge base, which are used by the inference engine in connection with the above facts to infer new knowledge. In particular, variable occurrences are classified as def, c-use or p-use according to their position in a statement and the kind of the statement. Similarly, global c-uses and nonlocal definitions are found according to their position within the segment. All this information consists of the new facts that are incorporated into the knowledge base. The testing strategy is described in the form of production rules. The facts and rules in the knowledge base are used to determine which pairs or paths are required by the given criterion. For this purpose two sets are constructed for each nonlocal definition found above. If the variable x has a nonlocal definition in node i, then dcu(x,i) is the set of all nodes j which have a global c-use of x, and there is a def-clear path with respect to x from i to j. The other set, dpu(x,i), contains all sets of nodes (j,j*), for every j* successor of j, where j has a p-use of x, and there is a def-clear path with respect to x from node i, in which the definition of x occurs, to j [29], [37]. This modification forces all branches to be taken

following a predicate use. In order to construct these sets, the inference engine uses the facts that represent, variable definitions, variable uses, and the flow of control. In this way, the subpaths $(i,n_k,\ldots,n_m,j)$ are obtained, where i contains a definition of a variable x, j a use of x, (if it is a p-use the successors j*s of j are also included), and $(n_k,\ldots,n_m)$ is a def-clear subpath with respect to x. Each of these subpaths is extended from its two ends, using the flow-of-control facts. There are several candidate extensions for each case because no unique testing set can be determined by these strategies unless some additional criteria are assumed [21]. We make these extensions for each end of every subpath in parallel and when the first of these reaches the start node, or the sink node respectively, then the process stops. In this way we obtain the shortest path from all those that include a required subpath. Of course, some of these paths may be untraversable, but since the problem is undecidable in general, it is natural to expect that the system selects unexecutable paths also [29]. Besides the automatically generated paths the user is able to specify his own statement sequence, basing this selection on his experience and/or on the module specification. The user can avoid the weaknesses of automatically generated path sets, especially infeasible paths. Some optimization could also be

carried out to arrive at a minimal set of control paths which cover all the test paths.

In the second step of the test data generation process the system generates the variable relationships that affect the program flow, i.e., the predicates for each of the selected paths. The predicates are initially expressed in terms of program variables. Since these variables, using assignment statements along the control path, can be expressed in terms of the input variables, the predicates can be re-written as constraints in terms of only the input variables. It begins to analyze the path from the source down to the sink segment. The path condition is initialized to the value true and the variables are initialized as follows: the input parameters are assigned symbolic names; variables that are initialized before execution are assigned their constant value; and all other variables are assigned the undefined value "?". After that each statement is interpreted as we visit each node on the path by substituting the current symbolic value of a variable wherever it is referenced. As we mentioned earlier, each node id and the corresponding statements of that node are represented as facts in the knowledge base. An input variable receives a new input value every time the variable receives an external value. For example, a variable in a read statement in a loop receives new value every time the loop is

executed. Expressions with constants are executed whenever possible. Whenever a conditional transfer of control is encountered, the branch predicate corresponding to the selected path is interpreted. When interpreting a branch predicate, the generated conditional expression provides a symbolic value for the branch predicate which is conjoined to the current path condition [7]. It is better that the evaluated branch predicate is first simplified and then conjoined to the previously generated path condition. There are several available algebraic manipulation systems that can be used to accomplish simplification, (for example, ALTRAN, a language for algebraic manipulations) [6]. Each time the path condition and the values for the variables are changed, they can be displayed to the user. Compound predicates are replaced by a series of simple predicates, so that hidden paths are not overlooked. Any constraint containing the OR operator is treated as a set of alternative constraints. If the first chosen constraint is inconsistent with the system of constraints, then the alternative is attempted. The AND operator, in compound predicates, is removed and each conjuctive term is made a separate constraint. There are a few transformation techniques that can be used to bring constraints in the required form. For example, if input variable $x_i$ is not constrained to be nonnegative, $x_i$ is replaced by $x_p - x_q$,

$x_p, x_q >= 0$, in the system of constraints. After the system of constraints is solved, by linear programming, for $x_p$, $x_q$, $x_i$ is computed by substituting back [6]. As mentioned earlier, only a subset of the paths in a program are executable. It is desirable to recognize nonexecutable paths as soon as possible in order to avoid worthless symbolic evaluation. Each constraint is examined for consistency with the condition of the partial path. If it is found to be consistent the branch predicate is conjoined to the path condition. If at any time during the analysis of the path the infeasibility of the path can be determined, a message is returned and the analysis of that path is terminated. If we cannot examine every time the consistency between a new constraint and the existing ones, due to limitations of operating system (storage limitations and interface problems), then this can be done at the end as a separate step, with the disadvantage of continuing the work for an infeasible path [6]. If an inconsistency between the constraints is discovered, an alternative branch may be taken in order to continue the analysis. It must take into account the corresponding def-use pair for this path, which must be covered again by the new alternative, and the constraint inconsistency discovered during the work on this path, which has been recorded in the knowledge base.

Cases in which all the constraints are linear can be solved using linear programming techniques. Input variables of real and integer data type can be handled by mixed linear programming algorithms. Logical data types are converted first: true to 1 and false to 0 [6]. We assume that there are no complex and double precision variables, because these can not be handled by the linear programming methods. For non-linear constraints, nonlinear programming techniques are required. Mathematical programming programs are usually very large and expensive to use for high dimensional problems [28]. Some other techniques exist that are partially successful. For example, Kuhn's backtrack search method, which produces solutions for some systems only [16]; and linearization, which linearizes the inequalities, solves the set of linearized inequalities, and then attempts a solution for the nonlinear set [23]. The conjugate gradient algorithm, ("hill climbing" procedure), appears to be better than other experimental methods, even though the procedure may not always terminate and needs human interaction [28]. When the constraints contain alphanumeric components - variables or literals - they cannot be processed by the inequality solver. Coward [8], proposed a solution to this problem. If only alphanumeric variables but no literals are present, then the constraints can be processed by an inequality solver which

treats all variables as numeric. Literals create a problem. The method creates a numeric token for each alphanumeric literal and passes the numeric token, along with the constants and variables, to the solver. Numeric tokens must give the same sort sequence as the literals they represent. For example, the tokens that represent "X" and "Y", must reflect the relationship "X" < "Y". Using linear programming or gradient hill-climbing algorithms a solution is provided when the consistency of inequalities is determined. When consistency or inconsistency cannot be determined, only the symbolic representations for a path can be provided.

At the end of this step the values of the input variables are the test data for the selected paths. The set of paths, through the whole process, has been divided into three sets: those for which it can generate test data, those for which it can determine infeasibility, and those for which it can do neither of the above. Since the test data selection problem is unsolvable this is the best to be expected. Hopefully, the last set is small most of the time, and it is possible with user intervention to resolve the ambiguity, i.e., for the paths in the set either test data are generated or their infeasibility is determined. Finally, the system determines for which paths, if any, required by the selection criterion it could not find test data and gives the test

coverage. For their corresponding def-use pairs, which were not exercised, it tries to find new paths to cover them. The knowledge base, as the whole process advances, is expanded with new information and this guides the selection of additional test data.

### 4.2.2 Software Reliability

As we mentioned in an earlier section, the time dependent models do not seem to be applicable during the unit testing phase, since the independent times between failures assumption is seriously violated [14]. Nelson's model is one of the principal domain-based models and it is attractive for programs of medium and small size. The model was developed from the basic properties of computer programs and uses probability theory with those aspects for which incomplete information is available, e.g., what input is chosen next. Where approximations are made, they are well-defined and the limits of their applicability are known [35].

The reliability of a module can be measured by running the program with a sample of n inputs and calculating R, the measured value, from the formula :

$$R = 1 - \frac{n_e}{n}$$

where $n_e$ is the number of inputs for which execution failures occur. In operational use, the inputs of n runs usually are not selected independently but in a definite sequence, such as ascending values of some input variable. Then if the n inputs in the sample are chosen from the input space at random according to the probability distribution that characterizes the operational requirement, the measured value of R is an unbiased estimate [35].

To measure R, the operational profile must be determined, i.e., the set $\{ p_i : i = 1,2,\ldots,n \}$, where each $p_i$ is a probability distribution and n is the number of inputs. There are numerous possible methods for sampling [31]. For example it is possible to perform sampling by test runs on selected partitions using a binomial or a beta distribution. It is possible way is to attach a cost to the length of the running time of some parts of the program and use stratified sampling with cost. In general the input domain may be partitioned by using the structure of the program or its specification, or both. Probabilities are assigned so that an input is chosen from each partition, based on an estimate of the occurrence of inputs in the operational use for which the reliability is being measured [35]. A sample of n inputs can be chosen at random, possibly with the aid of a random number generator, in accordance with the assigned probabilities. The

entire sample of n runs should be made without correcting any of the errors which caused the execution failures, and the estimation of R is calculated from the measurement data. The execution of the sampled test data may be either physical or symbolic. For this purpose we can use the symbolic evaluation part of our system to execute the test cases selected for the purpose of reliability testing.

### 4.2.3 Defect Prediction

The number of predicted defects in a module is computed from the formula given by Gaffney in [13]. The number of defects B in a module is related to the number of lines of code S of the module as follows :

$$B = 4.2 + 0.0015 ( S )^{4/3}$$

The above formula is used because it is a relatively simple relationship between the number of faults in the code and the number of lines of code, regardless of whether the code is written in a higher order language or not. As it was indicated in [13], the estimates based on the number of lines of code that comprise a module are as good or better than ones based on additional information. This means that knowledge of items such as the size of the vocabulary (operator and operand) used appears to be of little consequence to the

estimation of the number of defects beyond that based on the number of lines of code. A good estimation of the number of defects in a module can be obtained even though information such as vocabulary size is not available.

# CHAPTER V

# CONCLUSION

## 5.1 Summary

The design of an expert system, called SHELL, has been described and the first step of the design has been implemented. The system SHELL takes as input the flow graph of the module to be tested, and produces as output the number and definitions of test cases, the test coverage, the number of predicted defects, and the estimated reliability. The selection of the set of paths that must be tested is the first step, mentioned above, and is based on the theory developed by Rapps and Weyuker [29]. Specifically, the test data selection criterion all-uses is used, which asks that all interactions between a use (either c-use or p-use) and a definition that reaches it, be tested. This strategy was slightly modified to include branch testing, and it requires $O(n^2)$ test paths, where n is the number of nodes. The current limitation of the system SHELL is that the module to be tested uses only simple variables and four types of statements, assignment statements, conditional and unconditional transfer statements, and input and output statements.

## 5.2 Future Work

The plan for future work, in order to make the system SHELL complete, is to perform symbolic evaluation every time a path is derived. The system will analyze every path from the source down to the sink node, generating in this way the variable relationships that affect program flow, i.e., the predicates for each of the selected paths. The derived system of constraints is sent then to the inequality solver, which tries to generate test data or to determine the infeasibility of the path. In case it can do neither of the above, the user intervenes.

The simple programming language used here will be expanded to a more sophisticated one, including features such as arrays, dynamic allocation, aliasing, procedure and function calls, and recursion.

There are two main sources of information that are useful in testing, aside from the code itself. They are gathered knowledge about commonly occurring errors and input-output specifications (formal and informal). This information can be incorporated in the knowledge base to augment the approach used in the system SHELL for error detection. Examples include checking for errors in branch predicates, using even and odd values in an integer division by 2, using zero to

test divisions, checking for branching the wrong way on equality in predicates of the form X >= Y (by adding the case X = Y to the set of cases X >= Y and X < Y). On the other hand, a covered interaction by an executable path can be viewed as a partial computation, while the specifications provide information about what the program is intended to do. Then by combining the internal view, which is described by the interaction, with the external view, which is described in the specifications, conditions that need to be tested and likely sources of error can be discovered. For example the range of the variable, that is referenced, is defined in the specification. This information used in computation with the interaction would detect possible error.

It is hoped that the system SHELL, augmented with the work towards the directions mentioned above, will prove to be an effective, reasonable-cost aid in the selection of test data for programs written in the expanded programming language. It would improve productivity, quality and reliability in the software development process with subsequent gain being the reduced software cost.

# Appendix A

## A.1 The Program "SHELL"

```
/*   Operator  definitions   */


?-  op(60,    fx, nott).
?-  op(40,   xfx, equal).
?-  op(40,   xfx, noequal).
?-  op(40,   xfx, greater).
?-  op(40,   xfx, less).
?-  op(40,   xfx, eqgreat).
?-  op(40,   xfx, eqless).
?-  op(35,   xfx, gets).
?-  op(31,   yfx, or).
?-  op(31,   yfx, plus).
?-  op(31,   yfx, minus).
?-  op(21,   yfx, and).
?-  op(21,   yfx, times).
?-  op(21,   yfx, divint).
?-  op(21,   yfx, divreal).
?-  op(11,   xfx, mod).
?-  op(11,   xfy, power).
?-  op(10,    fx, uminus).


/*   Utility  routines   */

member(X, [X | _])   :- !.
member(X, [Y | Rest])   :-  member(X, Rest).

conc([], L, L).
conc([X | Rest1], List1, [X | Rest])   :-
            conc(Rest1, List1, Rest).

delete(X, [X | Rest], Rest)   :- !.
delete(X, [Y | Rest], [Y | Rest1])   :-
            delete(X, Rest, Rest1).

sublist(S, [L | Rest])   :-  conc(L1, L2, L),
                             conc(S, L3, L2), !.
sublist(S, [X | Rest])   :-  sublist(S, Rest).


/*************************************************************/
/*   Find  the  Variable  List  of each Statement   */
```

```
vars(X gets Exp, [X | Rest])  :-
            vars(Exp, Rest).

vars(Exp, Varlist)  :-
            ( Exp1  power    Exp2  =  Exp;
              Exp1  mod      Exp2  =  Exp;
              Exp1  times    Exp2  =  Exp;
              Exp1  divint   Exp2  =  Exp;
              Exp1  divreal  Exp2  =  Exp;
              Exp1  plus     Exp2  =  Exp;
              Exp1  minus    Exp2  =  Exp;
              Exp1  equal    Exp2  =  Exp;
              Exp1  noequal  Exp2  =  Exp;
              Exp1  less     Exp2  =  Exp;
              Exp1  greater  Exp2  =  Exp;
              Exp1  eqless   Exp2  =  Exp;
              Exp1  eqgreat  Exp2  =  Exp;
              Exp1  and      Exp2  =  Exp;
              Exp1  or       Exp2  =  Exp ),
            vars(Exp1, Temp1),
            vars(Exp2, Temp2),
            conc(Temp1, Temp2, Varlist).

vars(nott Exp, Varlist)  :-  vars(Exp, Varlist).

vars(uminus Exp, Varlist)  :-  vars(Exp, Varlist).

vars(X, Y)  :-
            numeric(X),  Y = [];
            atom(X),  Y = [X].


/*************************************************************/

/*   Process the statements in each block to find all
     nonlocal definitions, the global c_uses and the
     p_uses.                                                 */

processstmts(N, [First | Rest], Localdefs)  :-
            First = _ gets _,  vars(First, [Lvar | Rvars]),
            global_cusages(N, Rvars, Localdefs),
            check_rest(Lvar, Rest, Answer),
            ( not member(Lvar gets _, Rest),
              Answer = no,
              write('Nonlocal definition of var :  '),
              write(Lvar), nl,  asserta(nonlocal_def(N, Lvar));
              true ),
            processstmts(N, Rest, [Lvar | Localdefs]).
```

```
processstmts(N, [First | Rest], Localdefs)  :-
            First = write1(L),
            global_cusages(N, L, Localdefs),
            processstmts(N, Rest, Localdefs).

processstmts(N, [First | Rest], Localdefs)  :-
            First = read1(L),
            subprocess(N, L, Rest),
            conc(L, Localdefs, Nlocaldefs),
            processstmts(N, Rest, Nlocaldefs).

processstmts(N, [], _).

processstmts(N, [C], _)  :-
            C \= read1(_),  C \= write1(_),
            C \= _ gets _,  vars(C, Varlist),
            p_usages(N, Varlist).

subprocess(N, [Lvar | R], Rest)  :-
            check_rest(Lvar, Rest, Answer),
            ( not member(Lvar gets _, Rest),
              Answer = no,
              write('Nonlocal definition of var :  '),
              write(Lvar),  nl,
              asserta(nonlocal_def(N, Lvar));
              true ),
            subprocess(N, R, Rest).

subprocess(N, [], Rest).

check_rest(Lvar, [H | T], Answer)  :-
            H = read1(P),  member(Lvar, P),
            Answer = yes;
            check_rest(Lvar, T, Answer).

check_rest(Lvar, [], no).

global_cusages(N, [], Localdefs).

global_cusages(N, [Var | Rest], Localdefs)  :-
            member(Var, Localdefs), !,
            global_cusages(N, Rest, Localdefs).

global_cusages(N, [Var | Rest], Localdefs)  :-
            ( not global_c_use(N, Var),
              assertz(global_c_use(N, Var)),
              write('Global c-use of var :  '),
              write(Var),  nl;  true ),
```

```
            global_cusages(N, Rest, Localdefs).

p_usages(N,[]).

p_usages(N, [Var | Rest])  :-
            ( not p_use(N, Var),
              write('P-use of var :  '), write(Var), nl,
              assertz(p_use(N, Var)); true ),
              p_usages(N, Rest).


/*********************************************************/

/*   Start the main program by reading the file which
     contains the information provided by the flow graph
     in the form of lists.                                */

shell(File)  :-
            not openfile,  see(File),
            asserta(openfile),  tell(outdata),
            tab(25),  write(' Input  File '), nl,
            shell(File).

shell(File)  :-
            read(N),
            ( N \== end_of_file,  write(N),
             ( not blabel(_, source),  assertz(blabel(N, source)),
               assertz(blabel(s, sink)),  write('  * source * ');
               true ),
              read(Stmtlist),  assertz(statements(N, Stmtlist)),
              nl,  write(Stmtlist),  nl,
              processstmts(N, Stmtlist, []),
              read(Gotolist),  assertz(goto(N, Gotolist)),
             ( Gotolist == [],  retract(goto(N, Gotolist)),
               assertz(goto(N, [s])),  write('[s]'),
               write('  * sink *  ');  write(Gotolist)),
              nl,  write('.......__________.......'),  nl,
              shell(File);
              N == end_of_file,  assertz(executable([])),
              assertz(nonexecutable([])),  nl, nl,
              tab(25),  write(' PROCESSING -- phase 1'),
              nl, nl, nl,  seen,  shellc(proceede)).


/*********************************************************/

/*   For each non local definition of a variable X and
     a c_use (global) or a p_use of X, find a subpath
     from Bi to Bj def - free wrt X.                      */
```

```
shellc(_)  :-
            nonlocal_def(Bi, X),
            ( global_c_use(Bj, X);  p_use(Bj, X)),
            not explored_pair(Bi, Bj, X),
            drive_a_subpath(Bi, Bj, X),
            assertz(explored_pair(Bi, Bj, X)),
            shellc(continue);
            coverage(C, R),  ( C < R,  nl, nl,
            write('.....**************.....'),
            nl, nl, nl, tab(25),  write(' PROCESSING -- phase 2'),
            nl, nl, nl,  shell1(alt);
            assertz(required_pair(0, 0, x, 0)),
            rename,  shell3(end)).


/*   If such a subpath ( Bi --> Bj ) is found, check if it
     is already in the list of found executable paths, other-
     wise form a complete path covering that subpath and
     pass it to symbolic evaluator. If something goes wrong,
     check an alternative from Bi to Bj while the Source -->
     Bi and Bj --> Sink remain the same. If not an executable
     one is found try another pair wrt some Var.          */

drive_a_subpath(Bi, Bj, X)  :-
            path(Bi, Bj, bfs(X), Path2),  executable(L),
            name(X, [A | B]), nl,  write('^^^^^^^^^^^^^^^^^'), nl,
           ( global_c_use(Bj, X),
            ( not required_pair(Bi, Bj, X, 0),
              assertz(required_pair(Bi, Bj, X, 0)),
              write_req_pair(Bi, Bj, X, 0);  true ), nl,
             write('Def-clear subpath from '),  write(Bi),
             write(' to '),  write(Bj),  write(' wrt var '),
             put(A),  write(' : '),  nl,  write(Path2),  nl,
            ( [F | Rest] = Path2, ( sublist([F | Rest], L);
              sublist([yes(F) | Rest], L);
              sublist([no(F) | Rest], L)),  Answer = no,  I is 1;
              form_a_path(Bi, Bj, Bj, X, Path2, Answer), I is 0 ),
            ( Answer = no,  assertz(covered_pair(Bi, Bj, X, 0)),
              write_pair(Bi, Bj, X, 0, I);  true );
            ( p_use(Bj, X),  Answer = out,
             ( not required_pair(Bi, Bj, X, _),
               assertz(required_pair(Bi, Bj, X, 1)),
               assertz(required_pair(Bi, Bj, X, 2)),
               write_req_pair(Bi, Bj, X, 1),
               write_req_pair(Bi, Bj, X, 2);  true ),
              goto(Bj, [yes(Bk), no(Bq)]),
             ( not covered_pair(Bi, Bj, X, 1),
               conc(Path2, [yes(Bk)], Path21), nl,
```

```
                write('Def-clear subpath from '),  write(Bi) ,
                write(' to '),  write(Bj),  write(' wrt var '),
                put(A),  write(' - yes branch - : '),  nl,
                write(Path21),  nl,
               ( [F1 | Rest1] = Path21, ( sublist([F1 | Rest1], L);
                 sublist([yes(F1) | Rest1], L);
                 sublist([no(F1) | Rest1], L)),  Answer1 = no,
                                                    I1 is 1;
                 form_a_path(Bi, Bj, Bk, X, Path21, Answer1 ),
                                                    I1 is 0),
               ( Answer1 = no,  write_pair(Bi, Bj, X, 1, I1 ),
                 assertz(covered_pair(Bi, Bj, X, 1)); true );
              Answer1 = no ),
             ( not covered_pair(Bi, Bj, X, 2),
               conc(Path2, [no(Bq)], Path22),
               nl,  write('..............'),  nl, nl,
               write('Def-clear subpath from '),  write(Bi) ,
               write(' to '),  write(Bj),  write(' wrt var '),
               put(A),  write(' - no branch - : '),  nl,
               write(Path22),  nl,
              ( [F2 | Rest2] = Path22, (sublist([F2 | Rest 2], L);
                sublist([yes(F2) | Rest2], L);
                sublist([no(F2) | Rest2], L)),  Answer2 = no,
                                                   I2 is 1;
                form_a_path(Bi, Bj, Bq, X, Path22, Answer2 ),
                                                   I2 is 0 ),
              ( Answer2 = no,  write_pair(Bi, Bj, X, 2, I2 ),
                assertz(covered_pair(Bi, Bj, X, 2)); true );
             Answer2 = no))),
          (( Answer = yes,  drive_a_subpath(Bi, Bj, X);
                                    Answer = no );
           ( Answer1 = yes,  drive_a_subpath(Bi, Bj, X);
                                    Answer1 = no ),
           ( Answer2 = yes,  drive_a_subpath(Bi, Bj, X);
                                   Answer2 = no )).

drive_a_subpath(Bi, Bj, X).


/*   Expand a subpath covering a def - use pair, to reach
     the Source and Sink nodes                                  */

form_a_path(Bi, H, Bj, X, Path2, Answer)  :-
            blabel(Source, source),  blabel(Sink, sink),
            ( Bi == Source,  Path1 = [Bi];
              expand_from_source(Bi, Source, Path1)),
            delete(Bi, Path2, Path21),
```

```
            ( Bj == Sink,  Path3 = [];
              expand_to_sink(Bj, Sink, Path3)), nl,
              write('Expansions for the above subpath, '), nl,
             ( delete(Bi, Path1, Path11);
                          delete(no(Bi), Path1, Path11);
               delete(yes(Bi), Path1, Path11)),
             write('from source : '),  write(Path11),  nl,
             write('to sink     : '),  write(Path3),  nl,
             conc(Path1, Path21, L),  conc(L, Path3, Candpath),
            ( H == Bj,  S is 0;
             ( goto(H, [yes(Bj), _]),  S is 1;  S is 2)),
            write_candpath(Bi, S, H, Candpath, X),
                                              nonexecutable(NE),
            ( member(Candpath, NE),  Answer = yes, nl,
              write('The candidate path is a member of the '),
              write('nonexecutable paths : '), nl, write_list(NE);
              symb_eval(Candpath, Answer)).


/*   Pass the candidate path to symbolic evaluator and
     (simulating this step) we get an answer if it is exe-
     cutable or not.                                        */

symb_eval(Candpath, Answer)  :-
            tell(user),  write(Candpath),  nl,
            write('**  Dead ?'),  read(Answer), nl,
            write('.......... ************** ..........'), nl,
            tell(outdata),
            ( Answer = no,  retract(executable(Foundpaths)),
              assertz(executable([Candpath | Foundpaths])), nl ,
              write('The candidate path is executable.'),  nl;
              Answer = yes,  retract(nonexecutable(L)),
              assertz(nonexecutable([Candpath | L])), nl,
              write('The candidate path is nonexecutable,'), nl,
              write('try another alternative, - if any.'), nl) .

write_candpath(Bi, S, Bj, Candpath, X)  :-
            nl,  write('**  Candidate path to cover nodes'),
            write(' from '),  write(Bi),  write(' to '),
            write(Bj),  write(' wrt var '),  name(X, [F | R]),
            put(F),  ( S = 1,  write(', - branch yes -, ');
                       S = 2,  write(', - branch no -, ');
                       S = 0,  true ),
            write(' : '),  nl,  write(Candpath),  nl.

expand_from_source(Bi, Source, Path1)  :-
            known_paths(Source, Bi, unrestr, _, [Path1 | _]);
            path(Source, Bi, bfs, Path1).
```

```
expand_to_sink(Bj, Sink, Path3)  :-
            ( known_paths(Bj, Sink, unrestr, _, [Path31 | _]);
              path(Bj, Sink, bfs, Path31)),
            delete(Bj, Path31, Path3).

rename  :-
            retract(required_pair(Bi, Bj, X, S)), !,
            ( Bi == 0,  Bj == 0, !, true;
             ( not covered_pair(Bi, Bj, X, S),
               assertz(final_remained_pair(Bi, Bj, X, S));
               true ),
             assertz(required_pair(Bi, Bj, X, S)),
             rename ).

write_req_pair(Bi, Bj, X, S)  :-
            write('* Required pair : ['),  write(Bi),
            write(', '),  write(Bj),  write(']'),
            write(' wrt var '),  name(X, [F | R]),  put(F),
            ( S = 0,  write(', - c-use -.');
              S = 1,  write(', - p-use ---> branch yes -.');
              write(', - p-use ---> branch no -.')), nl, nl.

write_pair(Bi, Bj, X, S, I)  :-
            write('* Covered  pair : ['),  write(Bi),
            write(', '),  write(Bj),  write(']'),
            write(' wrt var '),  name(X, [F | R]),  put(F),
            ( I = 1, write(', by a known executable path');
              write(', by the candidate path')),
            ( S = 1,  write('- branch yes -');
              S = 2,  write('- branch no -');  true),  nl,
            ( I = 1, write(' from the list of found executable'),
              write(' paths :'), executable(L),  nl,
              write_list(L);  true ).

write_list([A | B])  :-
            write(A),  nl,  write_list(B).

write_list([])  :-
            nl.


/*   These three rules compute the coverage from the formula
     [ COVERED PAIRS BY PATHS ] / [ TOTAL REQUIRED PAIRS ].
     If it is >= 80% stop, else continue.                    */

coverage(C, R)  :-
            asserta(total_pairs(0)),
```

```
            assertz(required_pair(0,0,x,0)),  coverage1,
            asserta(covered_pairs(0)),
            assertz(covered_pair(0,0,x,0)),  coverage2,
            retract(covered_pairs(C)),
            retract(total_pairs(R1)),
            R is (R1 * 8) / 10.

coverage1  :-
            retract(required_pair(Bi, Bj, X, S)), !,
            ( Bi == 0, Bj == 0, !, true;
              retract(total_pairs(R1)),  N is R1 + 1,
              asserta(total_pairs(N)),
              assertz(required_pair(Bi, Bj, X, S)),
              coverage1).

coverage2  :-
              retract(covered_pair(Bi, Bj, X, S)), !,
              ( Bi == 0, Bj == 0, !, true;
                retract(covered_pairs(R1)),  N is R1 + 1,
                asserta(covered_pairs(N)),
                assertz(covered_pair(Bi, Bj, X, S)),
                coverage2).


/*************************************************************/

/*   If the coverage is < 80% try other alternatives from
     Bj to Sink with the previously chosen Source --> Bi,
     and combine with all Bi --> Bj if not executable. */

shell1(_)  :-
            required_pair(Bi, Bj, X, S),
            not remained_pair(Bi, Bj, X, S),
            not covered_pair(Bi, Bj, X, S),  K is 0,
            drive_alternative(Bi, Bj, X, S, K),
            ( retract(explored_pair(Bi, Bj, X));  true ),
             ( not covered_pair(Bi, Bj, X, S),
               assertz(remained_pair(Bi, Bj, X, S));  true ),
            blabel(Sink, sink),
            ( S = 0,  change_known_path(Bj, Sink);
              goto(Bj, [yes(Bk), no(Bq)]),
             ( S = 1,  change_known_path(Bk, Sink);
               S = 2,  change_known_path(Bq, Sink))),
            ( K \= 1,  shell1(continue);  shell3(end)).

drive_alternative(Bi, Bj, X, S, K)  :-
            blabel(Sink, sink),  blabel(Source, source),
```

```
                Bj \== Sink,
                known_paths(Source, Bi, unrestr, _, [Path1 | _]),
                ( S = 0,  ( derived_paths(Bj, Sink, 0, _, DP1),
                  derive_known_path(Bj, Sink, Path31, DP1);
                  path(Bj, Sink, bfs, Path31),
                  put_known_path(Bj, Sink, Path31)),
                  delete(Bj, Path31, Path3);
                  goto(Bj, [yes(Bk), no(Bq)]),
                 ( S = 1,  ( derived_paths(Bk, Sink, 0, _, DP1),
                   derive_known_path(Bk, Sink, Path31, DP1);
                   path(Bk, Sink, bfs, Path31),
                   put_known_path(Bk, Sink, Path31)),
                   delete(Bk, Path31, Path32),
                   conc([yes(Bk)], Path32, Path3);
                   S = 2,  ( derived_paths(Bq, Sink, 0, _, DP1),
                   derive_known_path(Bq, Sink, Path31, DP1);
                   path(Bq, Sink, bfs, Path31),
                   put_known_path(Bq, Sink, Path31)),
                   delete(Bq, Path31, Path32),
                   conc([no(Bq)], Path32, Path3))),  K is 0,
                known_paths(Bi, Bj, restr(X), _, L),  nl,
                write('^^^^^^^^^^^^^^^^^'), nl, nl,
                write_req_pair(Bi, Bj, X, S),
                write('Combine new alternative from '), write(Bj) ,
                write(' to sink : '),  conc([Bj], Path3, P),
                nl,  write(P), nl,  nl,
                write('with the old expansion from source to '),
                write(Bi),  write(' : '), nl,  write(Path1), nl, nl,
                write('and a member of known paths from '),
                write(Bi),  write(' to '),  write(Bj),
                write(' wrt var '), name(X, [F | R]), put(F),
                write(' : '),  nl,  write(L),
                form_alternative(Bi, Bj, X, S, K, L, Path1, Path3),
                ( K = 0,  drive_alternative(Bi, Bj, X, S, K),
                  true ).

drive_alternative(Bi, Bj, X, S, K).

form_alternative(Bi, Bj, X, S, K, L, L1, L3)  :-
                executable(E),  nonexecutable(NE),
                nl, nl,  write('.............'),
                next(Path21, L, W), nl, nl,
                write('Selected alternative from the above list :'),
                nl,  write(Path21),  nl,
                delete(Bi, Path21, Path2),
                ( S = 0,  Path22 = Path2;
                  goto(Bj, [yes(Bk), no(Bq)]),
```

```
                ( S = 1,  conc(Path2, [yes(Bk)], Path22);
                  conc(Path2, [no(Bq)], Path22))),
               ( conc([Bi], Path22, [F | Rest]),
                ( sublist([F | Rest], E);
                  sublist([yes(F) | Rest], E);
                  sublist([no(F) | Rest], E)),  Answer = no;
                 conc(L1, Path2, L2),  conc(L2, L3, Candpath),
                 write_candpath(Bi, S, Bj, Candpath, X),
                ( member(Candpath, NE),  Answer = yes, nl,
                  write('The candidate path is a member of the '),
                  write('nonexecutable paths :'), nl, write_list(NE);
                  symb_eval(Candpath, Answer))),
               ( Answer = yes,  ( W = [],  K is 0;
                 form_alternative(Bi, Bj, X, S, K, W, L1, L3));
                 Answer = no,  assertz(covered_pair(Bi, Bj, X, S)),
                 write_pair(Bi, Bj, X, S),
                 coverage(C, R),  ( C >= R,  K is 1;  K is 2 )).

form_alternative(Bi, Bj, X, S, K, L1, L3).

next(Path21, L, W)  :-
               member(Path21, L),
               delete(Path21, L, W).

derive_known_path(Bm, Bn, Path0, DP1)  :-
               next(Path0, DP1, DP),
               retract(derived_paths(Bm, Bn, 0, DP2, DP3)),
               assertz(derived_paths(Bm, Bn, 0, DP2, DP)).

put_known_path(Bm, Bn, Path0)  :-
               derived_paths(Bm, Bn, 1, _, DP),
               retract(derived_paths(Bm, Bn, 1, _, DP)),
               assertz(derived_paths(Bm, Bn, 1, [Path0 | DP],
                                                [Path0 | DP]));
               assertz(derived_paths(Bm, Bn, 1, [Path0], [Path0])).

change_known_path(Bm, Bn)  :-
               derived_paths(Bm, Bn, 1, DP, DP1),
               retract(derived_paths(Bm, Bn, 1, DP, DP1)),
               assertz(derived_paths(Bm, Bn, 0, DP, DP1));
               ( derived_paths(Bm, Bn, 0, DP, DP1),
                 retract(derived_paths(Bm, Bn, 0, DP, DP1)),
                 assertz(derived_paths(Bm, Bn, 0, DP, DP));  true).


/*************************************************************/
```

```
/*   If the achieved coverage is not >= 80% then continue
     with the final step, which generates the remaining
     alternatives Source --> Bi and combines them with the
     already generated Bi --> Bj, Bj --> Sink.          */

shell1(_)  :-
            not remained_pair(0, 0, x, 0),
            assertz(remained_pair(0, 0, x, 0)), nl,
            nl,  write('.....**********.....'),
            nl, nl, nl,  tab(25),
            write(' PROCESSING __ phase 3'), nl, nl, nl,
            shell2(final).

shell2(_)  :-
            remained_pair(Bi, Bj, X, S),
            ( Bi == 0,  Bj == 0,
              retract(remained_pair(Bi, Bj, X, S)),
              shell3(end);
             ( not covered_pair(Bi, Bj, X, S),  K is 0,
               drive_final_alt(Bi, Bj, X, S, K),
               ( not covered_pair(Bi, Bj, X, S),
                 retract(remained_pair(Bi, Bj, X, S)),
                 assertz(final_remained_pair(Bi, Bj, X, S));
                 true ),
            blabel(Source,source),  change_known_path(Source, Bi),
            ( K \= 1,  shell2(final);  shell3(end)))).

drive_final_alt(Bi, Bj, X, S, K)  :-
            blabel(Source, source),  Bi \== Source,
            ( derived_paths(Source, Bi, 0, _, DP),
              derive_known_path(Source, Bi, Path1, DP);
              path(Source, Bi, bfs, Path1), nl,
              put_known_path(Source, Bi, Path1)),
            write('^^^^^^^^^^^^^^^'), nl, nl,
            write_req_pair(Bi, Bj, X, S),
            write('Combine new alternative from source to '),
            write(Bi), nl,  write(Path1), nl, nl,
            known_paths(Bi, Bj, restr(X), _, L2),
            write('with a member of the known paths from '),
            write(Bi),  write(' to '),  write(Bj),
            write(' wrt var '),  name(X, [F | R]),
            put(F),  write(' :'), nl,  write(L2),
            drive_aux(Bi, Bj, X, S, K, Path1, L2),
            ( K = 0,  drive_final_alt(Bi, Bj, X, S, K);
              true ).

drive_final_alt(Bi, Bj, X, S, K).
```

```
drive_aux(Bi, Bj, X, S, K, Path1, L2)  :-
            blabel(Sink, sink),  next(Path21, L2, Y),
            nl, write('.....'), nl, nl,
            write('selected alternative from the first list :' ),
            nl,  write(Path21), nl,
            delete(Bi, Path21, Path22),
            ( S = 0,  Path2 = Path22,
              known_paths(Bj, Sink, unrestr, _, L3);
              goto(Bj, [yes(Bk), no(Bq)]),
             ( S = 1,  conc(Path22, [yes(Bk)], Path2),
               known_paths(Bk, Sink, unrestr, _, L3);
               conc(Path22, [no(Bq)], Path2),
               known_paths(Bq, Sink, unrestr, _, L3))), K is 0,
            nl, write('and a member of the known paths from ')),
            write(Bj),  write(' to sink : '), nl,  write(L3),
            form_final_alt(Bi, Bj, X, S, K, Path1, Path2, L3),
           ( K = 0,  ( Y = [],  drive_final_alt(Bi, Bj, X, S, K);
                        drive_aux(Bi, Bj, X, S, K, Path1, Y));
             true ).

drive_aux(Bi, Bj, X, S, K, Path1, L2).

form_final_alt(Bi, Bj, X, S, K, Path1, Path2, L3)  :-
            executable(E),  nonexecutable(NE),
            next(Path31, L3, W), nl,
            write('.....'), nl, nl,
            write('selected alternative
                                    from the second list : '),
            nl,  write(Path31), nl, nl,
            ( S = 0,  delete(Bj, Path31, Path3),  H = Bj;
              goto(Bj, [yes(Bk), no(Bq)]),
             ( S = 1,  delete(Bk, Path31, Path3),  H = Bk;
                       delete(Bq, Path31, Path3),  H = Bq)),
            ( conc([Bi], Path22, [F | Rest]),
             ( sublist([F | Rest], E);
               sublist([yes(F) | Rest], E);
               sublist([no(F) | Rest], E)),  Answer = no;
              conc(Path1, Path2, L1),
              conc(L1, Path3, Candpath),
              write_candpath(Bi, S, Bj, Candpath, X),
             ( member(Candpath, NE),  Answer = yes, nl,
               write('The candidate path is a member of the '),
               write('nonexecutable paths :'), nl,
               write_list (NE);
               symb_eval(Candpath, Answer))),
            ( Answer = yes,  ( W = [], K is 0;
              form_final_alt(Bi, Bj, X, S, K, Path1, Path2, W));
```

```
                Answer = no, retract(remained_pair(Bi, Bj, X, S)),
                write_pair(Bi, Bj, X, S),
                assertz(covered_pair(Bi, Bj, X, S)),
                coverage(C, R),  (C >= R, K is 1;  K is 2)).

form_final_alt(Bi, Bj, X, S, K, Path1, Path2, L3).


/*************************************************************/

shell3(End)  :-
              write('.............@.............'), nl, nl,
              nl, coverage(C, R),  nl,  T is (R * 10) / 8,
              ( C < R,  write('* No  More  Paths *'),
                nl;  true ), nl,
              write('**  Achieved  Coverage  :'),
              N is (C * 100) / T,  write(N),
              write('%'),  nl, nl,
              ( retract(remained_pair(0,0,x,0));  true),
              write('**  Remaining  Pairs  :   '),
              ( not final_remained_pair(Bi, Bj, _, _),
                write(' No More Pairs ');  remaining_pairs).

remaining_pairs  :-
              final_remained_pair(Bi, Bj, X, S),
              write(Bi),  write(' --> '),  write(Bj),
              ( S = 0,  write(',');
                S = 1,  write(', - branch yes -,');
                S = 2,  write(', - branch no -,')),
              write('  wrt var  '),  name(X, [F | R]),
              put(F),  write(' . '),  nl,  tab(26),
              fail;  true.


/*************************************************************/

/*   Compute a path between two nodes by breadth - first
     search. If pathtype is restricted, then it computes
     a def - clear path wrt some var X.                    */

path(Bi, Bj, Method, Path)  :-
              ( Method = bfs,  Pathtype = unrestr;
                Method = bfs(X),  Pathtype = restr(X)),
              not known_paths(Bi, Bj, Pathtype, _, _),
              assertz(known_paths(Bi, Bj, Pathtype, [[Bj]], [])),
              path(Bi, Bj, Method, Path).
```

```
path(Bi, Bj, Method, [Bi | Rest])  :-
              ( Method = bfs,  Pathtype = unrestr;
                Method = bfs(X),  Pathtype = restr(X)),
              known_paths(Bi, Bj, Pathtype, Ppaths, Paths),
              member([Bi | Rest], Ppaths),
              ( Method = bfs(_),  not ( Rest = [] );
                Method = bfs ),
              delete([Bi | Rest], Ppaths, Newpaths),
              retract(known_paths(Bi, Bj, Pathtype, _, _)),
              assertz(known_paths(Bi, Bj, Pathtype, Newpaths,
                                  [[Bi | Rest] | Paths])).

path(Bi, Bj, Method, Path)  :-
              ( Method = bfs,  Pathtype = unrestr;
                Method = bfs(X),  Pathtype = restr(X)),
              known_paths(Bi, Bj, Pathtype,
                                  [Ppath1 | Rest], Paths),
              expand(Bi, Method, [Ppath1 | Rest], Newpaths),
              retract(known_paths(Bi, Bj, Pathtype, _, _)),
              assertz(known_paths(Bi, Bj, Pathtype,
                                  Newpaths, Paths)),
              !,  Newpaths \= [],  path(Bi, Bj, Method, Path).

expand(Bi, Method, [Ppath1 | Rest], Newpaths)  :-
              expand_aux(Bi, Method, Ppath1, Temp1),
              expand(Bi, Method, Rest, Temp2),
              conc(Temp1, Temp2, Newpaths).

expand(Bi, Method, [], []).

expand_aux(Bi, Method, [Bj | Rest], Newpaths)  :-
              goto(Bk, _),
              legal_expansion(Bi, Bk, Method, [Bj | Rest],
                                                  Newfirst),
              assertz(queue([Bk, Newfirst | Rest])),
              fail;
              assertz(queue(bottom)),
              collect(Newpaths).

collect(L)  :-
              retract(queue(X)), !,
              ( X == bottom, !,  L = [];
                L = [X | Rest], collect(Rest)).

legal_expansion(Bi, Bk, Method, [Bj | Rest], Newfirst)  :-
              goto(Bk, L),
              ( member(Bj, L),  Newfirst = Bj;
```

```
    member(yes(Bj), L),  Newfirst = yes(Bj);
    member(no(Bj), L),  Newfirst = no(Bj)),
  ( Method = bfs(X),  Bk == Bi;
   ( not member(Bk, [Bj | Rest]),
     not member(yes(Bk), [Bj | Rest]),
     not member(no(Bk), [Bj | Rest]),
    ( Method = bfs(X),  Bk \== Bi,
      not nonlocal_def(Bk, X);
      Method = bfs ))).

/*************************************************************************/
```

## A.2 Documentation of "S H E L L"

**input**

The program uses as input the flow graph of the module to be tested. This information is stored in a file in the following form:

1. The id of the node, which is a number.
2. The statements contained in the node, are represented as the elements of a list, the statement list.
3. The successor(s) of the node, are represented as the elements of another list, the goto list. If it is a conditional node then it has two successors which are represented as relations, with the functor depending on the branch which the flow of control must follow in order to reach the corresponding node. The "yes" branch is always first in the goto list.

In case that more than one sink nodes exist, a new sink node is created, with empty statement and goto lists, and it becomes the successor of all sink nodes. This is in compliance with our assumption that each module has a single entry and a single exit.

The program starts to work by calling the rule "shell" with argument the name of the file containing the input.

**logic**

The program uses the test data selection criterion all-uses, from the family of criteria proposed by Rapps and Weyuker. It produces as output the set of paths required to test all interactions between a definition and a use, either a c-use or a p-use, reached by that definition. For each such pair, a subpath is generated from the node $B_i$ containing the definition, to the node $B_j$ with the use, which is def-clear with respect to corresponding variable, using breadth-first search. If $B_j$ is a conditional node with successors yes($B_k$) and no($B_q$) then both interactions, $B_i$ --> $B_j$, yes($B_k$) and $B_i$ --> $B_j$, no($B_q$), must be tested. Then this subpath is expanded to reach the Source and Sink nodes and this complete path is passed to symbolic evaluator, this step is simulated here.

If values for the variables, that will cause this path to be followed during execution, cannot be found, another alternative, if any, from $B_i$ to $B_j$ is tried while the parts of the path from Source to $B_i$ and from $B_j$ to Sink remain the same. When all such possibilities for all pairs have been tried and the coverage is not greater than or equal to 80%, the alternatives from $B_j$ to Sink are tried with all alternatives, already found, from $B_i$ to $B_j$ and the part of the path

from Source to $B_i$ remaining the same. The difference now is that every time a pair is covered, the program tests the coverage and if it is found to be greater than or equal to 80% it stops, otherwise it continues.

In the last step the alternatives from Source to $B_i$ are tried in combination with the already found alternatives from $B_i$ to $B_j$ and $B_j$ to Sink until the coverage is greater than or equal to 80% or no more alternatives exist to be tried. The order in which the alternatives are tried is as follows : Suppose from Source ---> $B_i$ two alternatives exist, [a, b], from $B_i$ ---> $B_j$ three def-clear subpaths with respect to X, [d, e, c], and from $B_i$ ---> Sink two alternatives, [f, g], then we have the order :

| | | |
|---|---|---|
| 1. [a, d, f] | 4. [a, d, g] | 7. [b, d, f] |
| 2. [a, e, f] | 5. [a, e, g] | 8. [b, d, g] |
| 3. [a, c, f] | 6. [a, c, g] | 9. [b, e, f] |
| | | 10. [b, e, g] |
| | | 11. [b, c, f] |
| | | 12. [b, c, g] |

In the two last steps (where the alternatives from $B_j$ to Sink and from Source to $B_i$ are tried) the system checks every time the coverage, and stops when it is found to be greater than or equal to 80%, which is a reasonable criterion for the termination of the process. The reason is that there

are time and cost limitations in program testing [15]. The number of alternatives may be very large, and symbolic evaluation is a time and space consuming process [7].

**output**

The program produces as output the required pairs of nodes, that need to be covered in order to satisfy the test data selection criterion, the list of executable paths which cover some or all pairs, the list of nonexecutable paths, the covered pairs, the remaining uncovered pairs and the achieved coverage.

**OPERATORS**

The operators used in the statements of the flow graph have different names than the usual ones, in order not to be confused with the built in operators of PROLOG, and they are redefined in the beginning of the program. Their relation to the standard operators is as follows:

```
nott    ---> not,  equal   ---> =  ,   noequal ---> \=,
greater ---> >  ,  less    ---> <  ,   eqgreat ---> >=,
eqless  ---> =< ,  gets    ---> := ,   or      ---> \/,
plus    ---> +  ,  minus   ---> -  ,   and     ---> /\,
times   ---> *  ,  divint  ---> div,   divreal ---> / ,
```

mod ---> mod, power ---> ** , uminus ---> -,(unary)
writel ---> write, readl ---> read.

## UTILITY ROUTINES

**member :**

Finds out whether or not an atom or a structure is among the elements of a list. Returns true or false.

**conc :**

Concatenates two lists. Returns a list which consists of the two input lists.

**delete :**

Deletes an element from a list. Returns the list with the corresponding element deleted.

**sublist :**

Checks whether or not a list is a sublist of some element of a list, which has lists as elements. Returns true or false.

## RULES

**vars**

It examines a particular statement of a node and extracts the variable occurrences, appearing in that statement, either definition or uses, constructing the variable list

of the statement. Returns a list of variables which in the case of an assignment statement has as first element a definition (and the rest are uses), otherwise it is consisting of uses (write statement) or definitions (read statement).

**processstmts**

These five rules take the statement list of each node in the flow graph and process every element of this list, with the help of some other rules that perform various tasks. We can have in the statement lists, assignment statements, input and output statements, and conditional transfer statements. In an assignment statement the variable occurrence in the left side is checked to see if it is a nonlocal definition, and if so it is asserted in the knowledge base. The same is done for every variable occurrence in the variable list of a read statement. The rest variable occurrences in an assignment statement (right side), and those in the list of a write statement are c-uses and are checked to see if they are global, in that case they are asserted in the knowledge base. Finally, the variable occurrences in a conditional statement are asserted as p-uses.

**global_c_usages**

Checks if there exists for a c-use a definition within the same block, because in such a case it is not global.

**p_usages**

Asserts the variable occurrence in a conditional statement as a p-use, if it is not already in the knowledge base for the same node id.

**shell**

It is the main rule of the program, this rule is invoked first, with argument the name of the file containing the input information. "Shell(File)" reads the information from the input file and asserts it in the knowledge base, separately for every node.

**shellc**

For each nonlocal definition of a variable and a use (either c-use or p-use) it will try to find a path to cover that pair. If no more pairs exist and the coverage is less than 80%, it calls the rule for the alternatives otherwise the terminating rule.

**drive_a_subpath**

This rule is called from "shellc" to examine whether or not there is a definition clear path with respect to variable X from node $B_i$, where a definition of X occurs, to node $B_j$, where a use of X occurs. If a path is found, by calling the "path" rules, it proceeds as follows: In the case that we

have a c-use in $B_j$, then it asserts this pair as required and checks if the path is a sublist of one of the already found executable paths, in which case the pair is already covered, otherwise it calls for path expansion to reach the Source and Sink nodes, i.e., a complete path, and then it is passed for symbolic evaluation. From there, an answer "yes" is returned if the path is not executable and a "no" if it is. If "yes", the pair is asserted as covered, otherwise the "drive_a_subpath" rule is called again to try an alternative subpath from $B_i$ to $B_j$ which has the same expansions reaching the Source and Sink nodes. This cycle is continued until the pair is covered by some executable path or no more alternatives exist to be tried. In both cases a true value is returned to rule "shellc" in order to continue with other pairs.

In the case that there is a p-use in node $B_j$, it does the same thing but now for two pairs [$B_i$, $B_j$, X, 1] and [$B_i$, $B_j$, X, 2], where "1" means to follow the "yes" branch and "2" means to follow the "no" branch. In other words, it examines the pairs $B_i$ ---> $B_j$, yes($B_k$) and $B_i$ ---> $B_j$, no($B_q$) separately, as above. The difference in these cases is that we are looking now for subpaths from $B_k$ or $B_q$ to Sink, not $B_j$ to Sink, and some adjustments are made to reflect this situation.

**form_a_path**

Given a path from $B_i$ to $B_j$, this rule is used to form the complete path from the Source to Sink, which covers the subpath from $B_i$ to $B_j$. First it checks if $B_i$ is the Source node or $B_j$ is the Sink node, in which cases we need no expansion. Otherwise it calls the rules for expansion, and concatenates these three subpaths in order to form the whole path. If the path is a member of the list of nonexecutable paths, already found, it needs no symbolic evaluation, otherwise it calls "symb_eval" rule to derive an answer.

**symb_eval**

This rule is a simulator for the symbolic evaluation and constraint solver steps. For the found candidate path it reads the answer "yes" or "no" from the user and passes it to the parent rule.

**write_candpath**

It prints a comment either to the user or in the file about the candidate path, i.e., which pair of nodes are to be covered, and which branch, "yes" or "no", in case of a p-use.

**expand_from_source**

If there is a known path in the knowledge base from

the Source to $B_i$ then return it, otherwise call the "path" rules to find one.

**expand_to_sink**

It performs exactly the same task but for a path from $B_j$ to Sink.

**rename**

This rule is an auxiliary routine to assert the pairs that are not covered (and required), as remained pairs in the knowledge base.

**write_req_pair**

This rule is called to print the required pair, when a def-clear subpath is found from the node containing the definition of a variable to the node containing the use of that variable. In case of a p-use it prints also the branch.

**write_pair**

Whenever a pair is covered by an existing executable path, this rule is invoked to print the ids of nodes, the variable, and the type of branch in case of a p-use.

**coverage**

This rule is used to invoke "coverage1", in order to compute the required pairs, and "coverage2" for the computa-

tion of the number of covered pairs. Finally, it computes the number which is equal to 80% of the totally required pairs.

**coverage1**

It computes, recursively, the total required pairs. Note that in the case of a p-use, we have two required pairs, "yes" and "no" branches.

**coverage2**

It computes, again recursively, the required pairs that have been covered by the time it is called.

**shell1**

In the previous section the program examined for all required pairs ($B_i$, $B_j$), if there exists a path from Source to Sink, which covers the definition-clear subpath from $B_i$ to $B_j$ with respect to some variable X, and it is executable. In such a case it asserted the pair covered, otherwise it continued with another alternative from $B_i$ to $B_j$, while the parts from Source to $B_i$ and from $B_j$ to Sink remain the same. Finally, if the pair ($B_i$, $B_j$) were not covered we have for that pair, in the list of known paths from $B_i$ to $B_j$, all the alternatives from $B_i$ to $B_j$ with respect to variable X.

In this section, it combines these alternatives with

the alternatives from $B_j$ to Sink, not yet tried, while the part from the Source to $B_i$ remains the same. Each time it checks the coverage, in order to stop further processing in the case that 80% has been achieved.

**drive_alternative**

Given are the pair ($B_i$, $B_j$), the variable X, and an indication of the kind of use of the variable X in node $B_j$. In the case of a p-use it also indicates the branch to be followed, "yes" or "no". If the node $B_j$ is the Sink there exist no more alternatives from $B_j$ to Sink and the rule returns the value true, in order to continue with another non-covered pair. Otherwise using the "path" rules find another alternative, if any, and call the rule "form_alternative" to examine the combinations of Source ---> $B_i$ (which remains the same all times), and $B_j$ ---> Sink (just found), with the elements of the list of the known paths from $B_i$ to $B_j$ with respect to variable X. Note that as in the "drive_a_subpath" rule, if we have a p-use in $B_j$ there are two branches to be followed, $B_j$ ---> yes($B_k$) and $B_j$ ---> no($B_q$). The variable S indicates which branch is examined currently, 1 for "yes" and 2 for "no", thus the list of known paths to be tried is not from $B_j$ to Sink but from $B_k$ or $B_q$ to Sink. Also all the necessary conversions are made by this rule to reflect the above mentioned difference.

Finally, if all combinations have been tried and the pair is still noncovered, it tries to find a new subpath from $B_j$ (or $B_k$, or $B_q$) to Sink and continues as above. This process stops when no more such subpaths exist, in which case it returns the value true in order to continue with another noncovered pair, or the pair is covered by some path.

**form_alternative**

In this rule the above combinations are formed, i.e., $path_1$ (Source ---> $B_i$), $path_3$ ($B_j$ [ $B_k$, $B_q$ respectively ] ---> Sink, and $path_2$ to be the elements of the list of known paths from $B_i$ ---> $B_j$. Every time a new element is combined, the rule checks whether or not it is a sublist of one of the found executable paths, i.e., the pair is covered, otherwise it forms the complete path and examines if it is a member of the list of nonexecutable paths, i.e., try other alternative. If none of the above is the case, the path is passed for symbolic evaluation. Depending on the answer it tries other alternatives or asserts the pair as covered. Every time a pair is covered, the coverage is computed in order to terminate this process if 80% of the required pairs have been covered by some executable path.

**next**

It is used as an auxiliary routine. Its function is

to delete the first element of a given list and to return it as its value.

**derive_known_path**

This rule is used to retrieve a known expansion of a subpath from the knowledge base. This expansion was found by the program for some other pair and was saved in order to avoid additional work.

**put_known_path**

It saves an expansion of a subpath, either from the source to the node containing the definition of a variable or from the node containing the use of the variable to the sink, in the knowledge base for later use.

**change_known_path**

This rule acts as a switch for the above two rules. If there is a known expansion in the knowledge base retrieve it, otherwise use the "path" rules to find one.

**shell2**

By calling this rule, we enter the final step of the process. If the current coverage is less than 80%, then for each remaining noncovered pair try to find alternative paths that may cover these pairs. Since for every pair ($B_i$, $B_j$) we already have all the alternatives from $B_i$ to $B_j$ and $B_j$ to

sink, the rule tries to find an alternative, if any, from the Source to $B_i$ and then to combine these three parts, in order to test the executability of the derived candidate path. Each time a pair is covered, it checks the coverage to decide whether or not the process will continue.

**drive_final_alt**

If node $B_i$ is the Source node then choose other pair, all alternatives for this pair have been tried, otherwise call the "path" rules to find a subpath from Source to $B_i$. It also derives the list of the alternatives from $B_i$ to $B_j$, and calls the next rule to continue. Note that if no more subpaths from Source to $B_i$ exist it returns the value true and continues with other noncovered pair. In the case that a subpath from Source to $B_i$ has been combined with all possible alternatives from $B_i$ to $B_j$ and from $B_j$ to Sink and the pair is still uncovered, the rule recursively calls itself to take another alternative, for the same pair, from Source to $B_i$, if any.

**drive_aux**

Depending on the kind of the uncovered pair, 0 is for c-use, 1 is for p-use ("yes" branch), and 2 is for p-use ("no" branch), this rule determines what alternatives of the third part of the path to derive. In other words, if the

indication is "0" then the alternatives are from $B_j$ to Sink, otherwise let's assume that the goto list for $B_j$ is [yes($B_k$), no($B_q$)]. In the case of "1" it must derive the alternatives from $B_k$ to Sink, otherwise from $B_q$ to Sink. The rule also makes all the necessary conversions to reflect each of the above situations.

At the end, two lists from alternative subpaths are available, from $B_i$ to $B_j$ (or one of $B_k$, $B_q$) and from $B_j$ (or one of $B_k$, $B_q$) to Sink, and a subpath from Source to $B_i$ ($path_1$). Then it calls the next rule to combine $path_1$ and a member of the first list with all elements of the other list. If nothing happens, the rule recursively calls it self to do the same thing with the rest members of the first list, until the pair is covered or no more alternatives exist, in which case it returns the value true to the parent rule in order to find another subpath from Source to $B_i$.

**form_final_alt**

The rule is used from "drive_aux" to form the candidate path. The parent rule passes to it as arguments two subpaths, one from Source to $B_i$ ($path_1$) and the other from $B_i$ to $B_j$ (or one of $B_k$, $B_q$ in the p-use cases) ($path_2$), and it tries to make all possible combinations of the above two with the elements of the list that consists of the subpaths from $B_j$ ($B_k$, $B_q$ respectively) to Sink.

First it checks if $path_2$ is a sublist of any of the found executable paths, i.e., the pair has been covered, and makes the value of the variable Answer to be "no". In this case asserts the pair as covered and returns control to the parent rule, to continue the processing of some other pair if the coverage is less than 80% or to call the terminating rule. Otherwise it forms the candidate path, by concatenating $path_1$, $path_2$, and the first element of the given list, and checks if it is a member of the list of the found non-executable paths. If this is the case the value of Answer becomes "yes", i.e., a recursive call to this rule will cause the same work to be done with some other member of the list with the alternatives for $path_3$, from $B_j$ ($B_k$, $B_q$ respectively) to Sink, if any, otherwise it returns control to the parent rule for continuation with some other alternative for $path_2$. If none of the above is the case, a new candidate path has been derived, it calls the "symb_eval" rule to process it. Depending on the value of the variable Answer that "symb_eval" returns, it proceeds exactly as explained above for these two cases, "yes" and "no".

**shell3**

It computes the final achieved coverage and checks whether or not it is greater than or equal to 80%. If it is

not so, i.e., there exist no more alternative paths to cover any of the remaining required pairs, the process is stopped. Then it uses the "remaining_pairs" rule to print all required pairs that were not covered.

**remaining_pairs**

Uses the knowledge base to retrieve and print the pairs that are required but not covered, if any.

**path**

There are three such rules which are used to compute a path between two nodes by breadth-first search. If the path type is to be restricted to some variable X, then it computes a definition-clear path with respect to the variable X. Now we will explain how this part works:

First the clause "known_paths(first node, last node, [[last node]],[ ])" is asserted in the knowledge base. The list of possible subpaths has one element, which in turn has one element, the last node, while the list of derived subpaths is empty. Then it uses the "expand" rule to make all possible one-node expansions and derives such as many as the number of predecessors of the last node. Before the rules continue to the next expansion steps, by recursive calls, the subpaths are examined to see whether or not one of them has reached the first node. If this is the case it places the

subpath in the list of derived subpaths, while the list of incompletely expanded subpaths may be used in case we need an alternative. Note also that after an expansion, if the list of expanded subpaths is empty it means that there exists no path from first node to last node and the process stops. Every change causes immediate update of the knowledge base.

**expand**

The rule is used to begin the expansion process for a list of subpaths. Using the "expand_aux" rule, it expands the head of the list and then, recursively, the tail of the list. Finally, it concatenates the lists of the expanded subpaths to form a new list. When the list to be expanded is empty, it returns the value true.

**expand_aux**

Given a subpath $[B_j, \ldots, B_q]$ it uses "legal_expansion" to expand the list of nodes and derive the new list $[B_k, B_j, \ldots, B_q]$ or $[B_k, yes(B_j), \ldots, B_q]$ or $[B_k, no(B_j), \ldots, B_q]$ and then asserts this new subpath in the knowledge base. The "fail" built-in function is used here and causes "expand_aux" to derive, in the same way, all the possible one-node expansions from node $B_j$.

**collect**

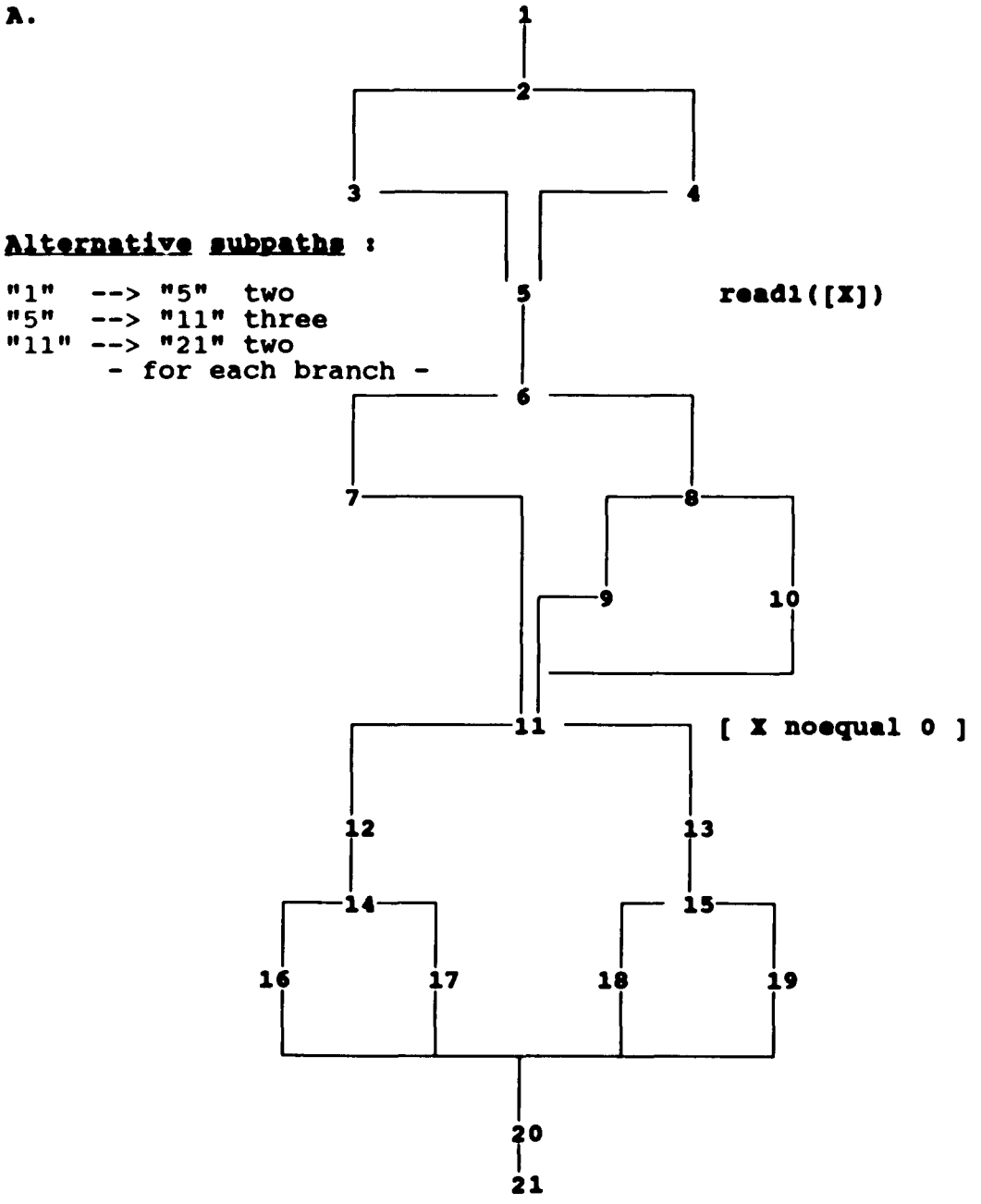
It retracts from the knowledge base all one-node expansions from a specified node and constructs a list that is consisting of all these subpaths.

**legal_expansion**

The rule is used to expand a subpath, represented as a list of nodes, by making the predecessor of the previously first node the current first node of the list. The rule checks if it is already in the list, to avoid cycles, and also takes into consideration that the expansion may be done through either a "yes" or a "no" branch, which must be reflected in the expanded list, and makes all the necessary conversions. Also, if the method used is the breadth-first search with respect to a variable, i.e., a definition-clear path with respect to that variable is needed, it discards the choices of such nodes that contain a definition of the variable.

## Appendix B

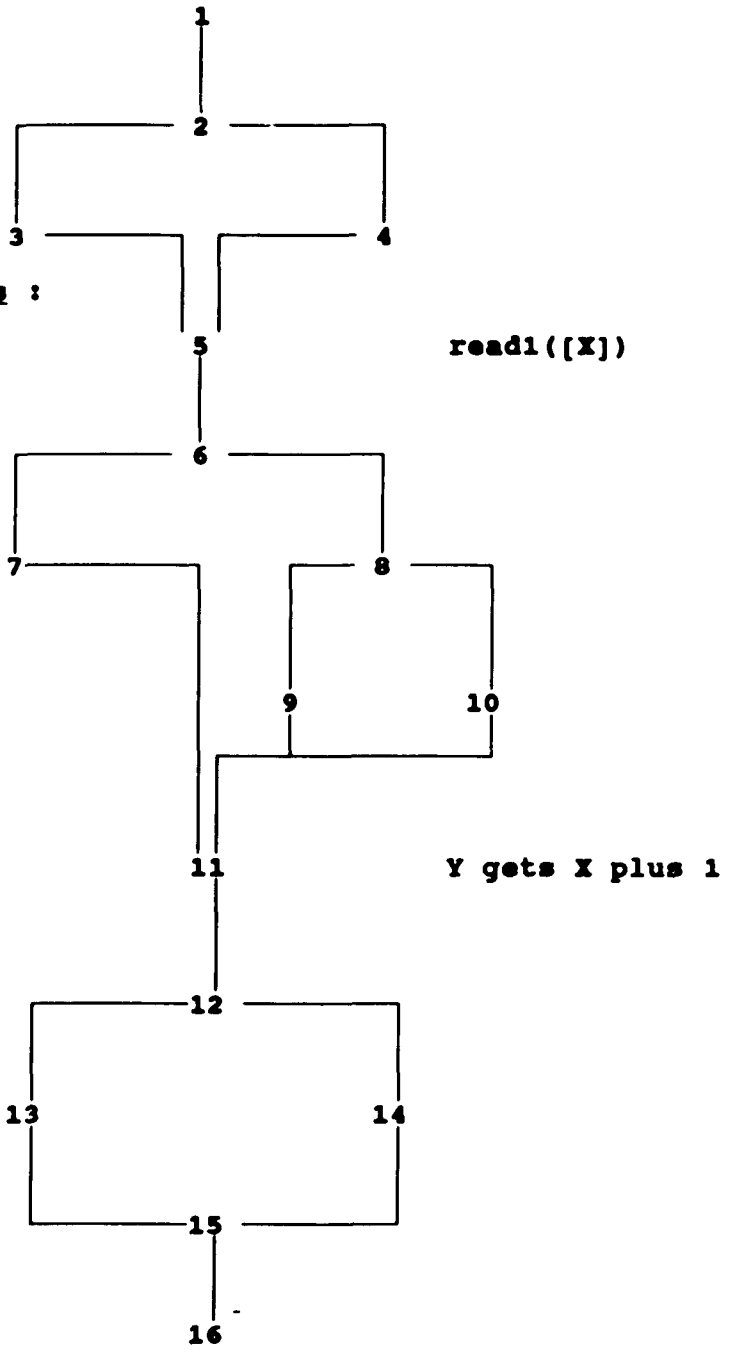
A.




**Alternative subpaths :**

"1" --> "5" two
"5" --> "11" three
"11" --> "21" two
- for each branch -

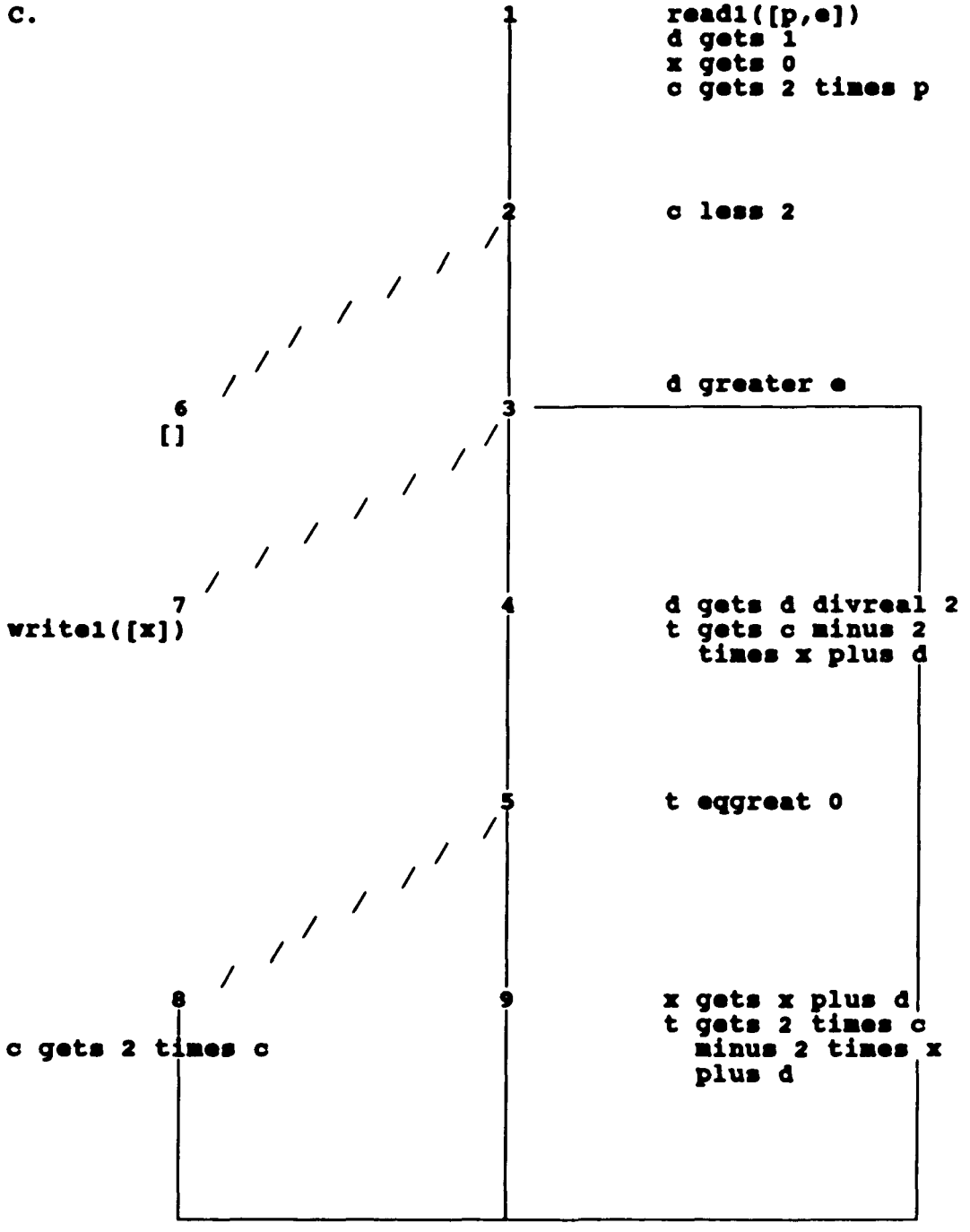
B.




**Alternative subpaths :**

"1" --> "5" two
"5" --> "11" three
"11" --> "16" two

c.


1
read1([p,e])
d gets 1
x gets 0
c gets 2 times p
2
c less 2
6
[]
3
d greater e
7
write1([x])
4
d gets d divreal 2
t gets c minus 2
times x plus d
5
t eqgreat 0
8
c gets 2 times c
9
x gets x plus d
t gets 2 times c
minus 2 times x
plus d

## B.2 Walking The Input Through The Program "SHELL"

We examine the processing of three sample input programs to demonstrate how the program "SHELL" handles the various cases. Each of the first two input files (see their flow graphs in pages 85 and 86), has only one definition of a variable X, in the statement "read1([X])", and a use of that variable. This use of variable X is a p-use for the first case (in the statement "X noequal 0"), and a c-use for the second case (in the statement "Y gets X plus 1"). The reason we choose such flow graphs is that many times the paths corresponding to some def-use pairs are covered either by already found executable paths for other pairs or by known nonexecutable paths, and if the coverage is greater than or equal to 80%, the processing stops at this point. Thus we don't know how some parts of the program "SHELL" work, and what output is produced in these cases. By designing the flow graphs to have alternative subpaths from the start node to the node containing the definition of the variable, from there to the node containing the use of that variable, and finally from there to the sink node, we cause the program to reach its final step, assuming that every found path is not executable.

The third input file is the flow graph of a module with different kinds of statements and def-use pairs, and it was taken from [29].

## A. P = USE CASE

The program begins by reading the input file, in rules "shell(File)", which consists of the ids of the nodes in the flow graph and the statement and goto lists of each node, and then print this information to the output file. If there is more than one sink node in the flow graph it creates a new sink node and updates the goto lists of all sink nodes so that they contain the id "s" of the new sink node. Then the program "shell" (using other rules), processes each statement list to find the nonlocal definitions, the global c-uses, and the p-uses of all existing variables, and prints this information along with labels for the source and the sink nodes in order to distinguish them from the rest nodes. Each time new information is extracted, the knowledge base is updated to reflect the changes and to be used in later processing. In this particular example the program "shell" finds that "1" is the source node, "21" is the sink node, so makes "s" to be the new sink node and puts the id "s" in the goto list of "21". Also it prints that the definition of X in the

statement "read1([X])" in node "5" is a nonlocal one and the use of X in the statement "X noequal 0" in node "11" is a p-use of the variable X.

After finishing with these details the program enters the PROCESSING -- phase 1. In this phase it checks to find the required def-use pairs, i.e., those pairs for which a def-clear subpath exists with respect to some variable X which occurs in the definition, from the node containing the definition to the node containing the use of variable X. If it is a p-use, then both outgoing branches of the node that contains the p-use must be tested, thus two def-use pairs are required. Returning to our example, the program finds that two pairs are required: [5,11] with respect to variable X, branch "yes", and [5, 11] with respect to variable X, branch "no". For the first required pair the corresponding def-clear subpath with respect to the variable X is: [ 5, 6, yes(7), 11, yes(12) ]. The next step is to check if this subpath is a sublist of some path in the list of found executable paths, i.e., the pair has been covered by a found executable path. Because this is not the case here, the program expands the above subpath to reach the source and sink nodes, thus deriving the candidate path : [1, 2, yes(3), 5, 6, yes(7), 11, yes(12), 14, yes(16), 20, 21, s ]. Before it is passed to symbolic evaluation, it is checked to see if

it is a member of the list of found nonexecutable paths, thus avoiding additional work. Note that we want to go through all steps, thus we suppose that the answer " nonexecutable path " is returned every time from the symbolic evaluator. Therefore, the program must search for another alternative subpath from the node containing the definition to the node containing the use, which will be combined with the same expansions found above. But before proceeding further, it has to do first the same work for the pair [5, 11] - branch "no". For the required pair [5,11] with respect to variable X, "no" branch, it finds the def-clear subpath [ 5, 6, yes(7), 11, no(13) ] which is not a sublist of any of the executable paths, currently this list is empty, therefore it is expanded to reach the source and sink nodes. The derived path is: [ 1, 2, yes(3), 5, 6, yes(7), 11, no(13), 15, yes(18), 20, 21, s ], and is not a member of the list of nonexecutable paths. By assuming that it is nonexecutable the program must search for another alternative (same work as in the branch "yes" case).

This processing is continued by trying all remaining alternatives, from the node containing the definition to the node containing the use, for both branches - "yes" and "no". These alternatives are as follow:

1. For "yes" branch: [5, 6, no(8), yes(9), 11, yes(12)],

[5, 6, no(8), no(10), 11, yes(12)].

2. For "no" branch: [5, 6, no(8), yes(9), 11, no(13)],
[5, 6, no(8), no(10), 11, no(13)].

Note that the expansions for the subpaths belonging in the same category, remain unaltered. For the category (1) we get [1, 2, yes(3)] and [14, yes(16), 20, 21, s], while for the category (2) we have [1, 2, yes(3)] and [15, yes(18), 20, 21, s]. Since we assumed for this example that all paths are not executable, after this work is done the coverage is less than 80 %, and the program enters the PROCESSING -- phase 2.

At this point we know all the alternatives, for every required pair that is still noncovered by some executable path, from the node containing the definition to the node containing the use. So we use now a member of the list of these alternatives for each noncovered pair in combination with a new expansion for this subpath from the node in which the use occurs to the sink, and the old expansion, used in phase 1, from the source to the node with the definition of the variable. If the path derived in this way is not executable, then the program tries another member of the list of alternatives with the same expansions. In our example, the first noncovered pair to be tried is [5, 11] with respect to variable X, branch "yes", has a new expansion from node "11" to sink node: [ 11, yes(12), 14, no(17), 20, 21, s ]

and a list of three alternatives from node "5" to node "11", while the old expansion from the source to node "5" is [1, 2, yes(3), 5]. Supposing that all paths are nonexecutable, in order to drive the program through all steps, we derive three candidate paths which are placed, as nonexecutable ones, into the list of nonexecutable paths. Then the second pair is examined, [5,11] with respect to variable X, "no" branch, in exactly the same way, and then the program enters the PROCESSING -- phase 3. Note that since the expansions of a subpath are independent of the variable, every time we use an expansion we can derive it from the knowledge base, if this expansion were previously found for some other pair. Also, every time a pair is covered, the coverage is calculated in order to terminate the process in case that it is equal to or greater than 80 %.

In this last phase of the processing, for each pair that is noncovered yet, we have in the knowledge base the lists of alternative subpaths, from the node containing the definition of the variable to the node containing the use of that variable and from there to the sink node. Thus "SHELL" tries now a path for each pair which consists of one member from each of the above lists and a new expansion, which may has been already found for some other pair and placed into the knowledge base, from the source to the node containing

the definition of the variable. If the candidate path, which was derived in this way, is nonexecutable then other members of the lists are combined with the expansion from the source node to the node containing the definition of the variable. Again, every time a pair is covered by some executable path the coverage is computed to see if further processing is not needed. In this particular example, the first required pair [5, 11] with respect to variable X, branch "yes", has a new expansion from source to "5": [ 1, 2, no(4), 5 ] and the lists of alternatives are: [ [5, 6, no(8), no(10), 11], [5, 6, no(8), yes(9), 11], [5, 6, yes(7), 11] ] from "5" to "11" and [ [12, 14, no(17), 20, 21, s], [12, 14, yes(16), 20, 21, s] ] from "11" to sink. Thus, supposing again that all found candidate paths are nonexecutable, the program tries six candidate paths for this pair (1 x 3 x 2 = 6), and places them into the list of nonexecutable paths. Then it continues with the next noncovered pair [5, 11] with respect to variable X, branch "no", which has similar processing. When no more paths can be found for the noncovered pairs, it prints the coverage and the remaining noncovered pairs.

## B. C - USE CASE

This example is the same as above, with the exception that there is only a c-use of a variable X in node "11", in the statement "Y gets X plus 1". Again here we assume that every found candidate path is nonexecutable, and this drives the program "SHELL" through all steps for the case of a c-use. The processing in this example is exactly the same as that in example A. Note that we have here two alternatives from the source node to the node containing the definition, three alternatives from there to the node containing the use, and two alternatives from there to the sink. Thus the number of the paths that our program finds, supposing that all of them are nonexecutable, is 2 x 3 x 2 = 12.

## c. MIXED CASE = 1

As in the previous examples, the program reads in the input file, and after processing the statement list of each node, prints the flow graph, specifying also the nonlocal definitions, global c-uses and p-uses in every node, the sink and source nodes, and enters the PROCESSING -- phase 1.

It finds for the first required pair [9,4] with respect to variable X, c-use, the def-clear path [9, 3, yes(4)] and expands it, thus deriving the candidate path [1, 2, yes(3), yes(4), 5, yes(9), 3, yes(4), 5, no(8), 3, no(7), s ], which we suppose is nonexecutable, and places it into the list of nonexecutable paths. Since there is no other alternative subpath from "9" to "4" to be tried, it continues this step with some other pair. The next pairs [9,7] with respect to variable X, c-use, and [[9,9] with respect to variable X, c-use, are also supposed to be nonexecutable so the processing is exactly the same. For the pair [8,4], c-use, with respect to variable C, we suppose its executability, so the program prints it as covered and places the corresponding path into the list of executable paths. Note that whenever a def-clear subpath is found for a pair it is checked to see if it is a sublist of a member in the list of found executable paths, while a candidate path is checked, before it is passed for symbolic evaluation, to see if it is a member of the list of

found nonexecutable paths, in order to avoid additional work. This is the case for the next pair [8,8], c-use, with respect to variable C, covered by a known executable path, and for the last one [1,3] with respect to variable E, p-use - branch "no", whose found candidate path is a member of the current list of nonexecutable paths. The path for the pair [8,9], c-use, with respect to variable C, is supposed to be executable and the program continues with the pair [4,5] with respect to variable T, p-use, which has both of its branches covered by some known executable paths. Each time a pair is not covered by some executable path, an alternative is tried, if there exists one, while the expansions remain the same.

When all pairs have been tried, and before the program enters the PROCESSING -- phase 2, the coverage is computed, (here 20 pairs out of total 25 have been covered). So the coverage is 80 % and there is no need for additional work. The program prints the coverage and the remaining noncovered pairs and terminates.

C. **MIXED CASE = 2**

This example does not need any comment, because it is the same as above with the exception that all candidate paths are supposed to be executable. Thus a pair is covered either by the candidate path or by some path in the list of known executable paths. At the end the printed coverage is 100 % and there are no more remaining pairs.

Input File

```
1  * source *
[]
[2]
.......__________.......
2
[]
[yes(3),no(4)]
.......__________.......
3
[]
[5]
.......__________.......
4
[]
[5]
.......__________.......
5
[read1([x])]
Nonlocal definition of var :  x
[6]
.......__________.......
6
[]
[yes(7),no(8)]
.......__________.......
7
[]
[11]
.......__________.......
8
[]
[yes(9),no(10)]
.......__________.......
9
[]
[11]
.......__________.......
10
[]
[11]
.......__________.......
11
[(x noequal 0)]
P-use of var :  x
[yes(12),no(13)]
.......__________.......
12
[]
[14]
.......__________.......
13
[]
[15]
.......__________.......
14
[]
[yes(16),no(17)]
.......__________.......
15
```

```
[]
(yes(18),no(19))]
.......__________.......
16
[]
[20]
.......__________.......
17
[]
[20]
.......__________.......
18
[]
[20]
.......__________.......
19
[]
[20]
.......__________.......
20
[]
[21]
.......__________.......
21
[]
[s]  * sink *
.......__________.......
```

PROCESSING -- phase 1

```
..............
* Required pair : [5, 11] wrt var x, - p-use ---> branch yes -.

* Required pair : [5, 11] wrt var x, - p-use ---> branch no -.


Def-clear subpath from 5 to 11 wrt var x - yes branch - :
[5,6,yes(7),11,yes(12)]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [14,yes(16),20,21,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,yes(3),5,6,yes(7),11,yes(12),14,yes(16),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

Def-clear subpath from 5 to 11 wrt var x - no branch - :
[5,6,yes(7),11,no(13)]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [15,yes(18),20,21,s]
```

```
**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,yes(3),5,6,yes(7),11,no(13),15,yes(18),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

...............

Def-clear subpath from 5 to 11 wrt var x - yes branch - :
[5,6,no(8),yes(9),11,yes(12)]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [14,yes(16),20,21,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,yes(3),5,6,no(8),yes(9),11,yes(12),14,yes(16),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

Def-clear subpath from 5 to 11 wrt var x - no branch - :
[5,6,no(8),yes(9),11,no(13)]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [15,yes(18),20,21,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,yes(3),5,6,no(8),yes(9),11,no(13),15,yes(18),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

...............

Def-clear subpath from 5 to 11 wrt var x - yes branch - :
[5,6,no(8),no(10),11,yes(12)]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [14,yes(16),20,21,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,yes(3),5,6,no(8),no(10),11,yes(12),14,yes(16),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

Def-clear subpath from 5 to 11 wrt var x - no branch - :
[5,6,no(8),no(10),11,no(13)]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [15,yes(18),20,21,s]
```

```
** Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,yes(3),5,6,no(8),no(10),11,no(13),15,yes(18),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.........***************.........

                        PROCESSING -- phase 2


...............

* Required pair : [5, 11] wrt var x, - p-use ---> branch yes -.

Combine new alternative from 11 to sink :
[11,yes(12),14,no(17),20,21,s]

with the old expansion from source to 5 :
[1,2,yes(3),5]

and a member of known paths from 5 to 11 wrt var x :
[[5,6,no(8),no(10),11],[5,6,no(8),yes(9),11],[5,6,yes(7),11]]

.............

Selected alternative from the above list :
[5,6,no(8),no(10),11]

** Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,yes(3),5,6,no(8),no(10),11,yes(12),14,no(17),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.............

Selected alternative from the above list :
[5,6,no(8),yes(9),11]

** Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,yes(3),5,6,no(8),yes(9),11,yes(12),14,no(17),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.............

Selected alternative from the above list :
[5,6,yes(7),11]

** Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,yes(3),5,6,yes(7),11,yes(12),14,no(17),20,21,s]
```

```
The candidate path is nonexecutable,
try another alternative, - if any.

...............

* Required pair : [5, 11] wrt var x, - p-use ---> branch no -.

Combine new alternative from 11 to sink :
[11,no(13),15,no(19),20,21,s]

with the old expansion from source to 5 :
[1,2,yes(3),5]

and a member of known paths from 5 to 11 wrt var x :
[[5,6,no(8),no(10),11],[5,6,no(8),yes(9),11],[5,6,yes(7),11]]

.............

Selected alternative from the above list :
[5,6,no(8),no(10),11]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,yes(3),5,6,no(8),no(10),11,no(13),15,no(19),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.............

Selected alternative from the above list :
[5,6,no(8),yes(9),11]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,yes(3),5,6,no(8),yes(9),11,no(13),15,no(19),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.............

Selected alternative from the above list :
[5,6,yes(7),11]

**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,yes(3),5,6,yes(7),11,no(13),15,no(19),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

.........***************.........

                    PROCESSING __ phase 3


...............
```

```
* Required pair : [5, 11] wrt var x, - p-use ---> branch yes -.

Combine new alternative from source to 5
[1,2,no(4),5]

with a member of the known paths from 5 to 11 wrt var x :
[[5,6,no(8),no(10),11],[5,6,no(8),yes(9),11],[5,6,yes(7),11]]
'''''''''''''''

selected alternative from the first list :
[5,6,no(8),no(10),11]

and a member of the known paths from 11 to sink :
[[12,14,no(17),20,21,s],[12,14,yes(16),20,21,s]]
..............

selected alternative from the second list :
[12,14,no(17),20,21,s]


** Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -, :
[1,2,no(4),5,6,no(8),no(10),11,yes(12),14,no(17),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[12,14,yes(16),20,21,s]


** Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -, :
[1,2,no(4),5,6,no(8),no(10),11,yes(12),14,yes(16),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

'''''''''''''''

selected alternative from the first list :
[5,6,no(8),yes(9),11]

and a member of the known paths from 11 to sink :
[[12,14,no(17),20,21,s],[12,14,yes(16),20,21,s]]
..............

selected alternative from the second list :
[12,14,no(17),20,21,s]


** Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -, :
[1,2,no(4),5,6,no(8),yes(9),11,yes(12),14,no(17),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............
```

```
selected alternative from the second list :
(12,14,yes(16),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,no(4),5,6,no(8),yes(9),11,yes(12),14,yes(16),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

''''''''''''''''

selected alternative from the first list :
[5,6,yes(7),11]

and a member of the known paths from 11 to sink :
[[12,14,no(17),20,21,s],[12,14,yes(16),20,21,s]]
..............

selected alternative from the second list :
[12,14,no(17),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,no(4),5,6,yes(7),11,yes(12),14,no(17),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[12,14,yes(16),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch yes -,  :
[1,2,no(4),5,6,yes(7),11,yes(12),14,yes(16),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.
^^^^^^^^^^^^^^^

* Required pair : [5, 11] wrt var x, - p-use ---> branch no -.

Combine new alternative from source to 5
[1,2,no(4),5]

with a member of the known paths from 5 to 11 wrt var x :
[[5,6,no(8),no(10),11],[5,6,no(8),yes(9),11],[5,6,yes(7),11]]
''''''''''''''''

selected alternative from the first list :
[5,6,no(8),no(10),11]

and a member of the known paths from 11 to sink :
[[13,15,no(19),20,21,s],[13,15,yes(18),20,21,s]]
..............

selected alternative from the second list :
[13,15,no(19),20,21,s]
```

```
**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,no(4),5,6,no(8),no(10),11,no(13),15,no(19),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[13,15,yes(18),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,no(4),5,6,no(8),no(10),11,no(13),15,yes(18),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the first list :
[5,6,no(8),yes(9),11]

and a member of the known paths from 11 to sink :
[[13,15,no(19),20,21,s],[13,15,yes(18),20,21,s]]
..............

selected alternative from the second list :
[13,15,no(19),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,no(4),5,6,no(8),yes(9),11,no(13),15,no(19),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[13,15,yes(18),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,no(4),5,6,no(8),yes(9),11,no(13),15,yes(18),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the first list :
[5,6,yes(7),11]

and a member of the known paths from 11 to sink :
[[13,15,no(19),20,21,s],[13,15,yes(18),20,21,s]]
..............
```

```
selected alternative from the second list :
[13,15,no(19),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,no(4),5,6,yes(7),11,no(13),15,no(19),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.

.............

selected alternative from the second list :
[13,15,yes(18),20,21,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x, - branch no -,  :
[1,2,no(4),5,6,yes(7),11,no(13),15,yes(18),20,21,s]

The candidate path is nonexecutable,
try another alternative, - if any.
.............@.............


* No  More  Paths *

**  Achieved  Coverage  :0.0000000000E+00%

**  Remaining  Pairs  :   5 --> 11, - branch yes -,  wrt var  x .
                          5 --> 11, - branch no -,  wrt var  x .
```

Input File

```
1  * source *
[]
[2]
.......__________.......
2
[]
[yes(3),no(4)]
.......__________.......
3
[]
[5]
.......__________.......
4
[]
[5]
.......__________.......
5
[read1([x])]
Nonlocal definition of var :  x
[6]
.......__________.......
6
[]
[yes(7),no(8)]
.......__________.......
7
[]
[11]
.......__________.......
8
[]
[yes(9),no(10)]
.......__________.......
9
[]
[11]
.......__________.......
10
[]
[11]
.......__________.......
11
[(y gets(x plus 1))]
Global c-use of var :  x
Nonlocal definition of var :  y
[12]
.......__________.......
12
[]
[yes(13),no(14)]
.......__________.......
13
[]
[15]
.......__________.......
14
[]
[15]
.......__________.......
```

```
15
( )
[16]
.......__________.......
16
[ ]
[s]  * sink *
.......__________.......
```

PROCESSING -- phase 1

```
---------------
* Required pair : [5, 11] wrt var x, - c-use -.

Def-clear subpath from 5 to 11 wrt var x :
[5,6,yes(7),11]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [12,yes(13),15,16,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,yes(3),5,6,yes(7),11,12,yes(13),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.

---------------

Def-clear subpath from 5 to 11 wrt var x :
[5,6,no(8),yes(9),11]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [12,yes(13),15,16,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,yes(3),5,6,no(8),yes(9),11,12,yes(13),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.

---------------

Def-clear subpath from 5 to 11 wrt var x :
[5,6,no(8),no(10),11]

Expansions for the above subpath,
from source : [1,2,yes(3)]
to sink     : [12,yes(13),15,16,s]

**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,yes(3),5,6,no(8),no(10),11,12,yes(13),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.
```

```
.....**************.....

                    PROCESSING -- phase 2


..............

* Required pair : [5, 11] wrt var x, - c-use -.

Combine new alternative from 11 to sink :
[11,12,no(14),15,16,s]

with the old expansion from source to 5 :
[1,2,yes(3),5]

and a member of known paths from 5 to 11 wrt var x :
[[5,6,no(8),no(10),11],[5,6,no(8),yes(9),11],[5,6,yes(7),11]]

.............

Selected alternative from the above list :
[5,6,no(8),no(10),11]

**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,yes(3),5,6,no(8),no(10),11,12,no(14),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.............

Selected alternative from the above list :
[5,6,no(8),yes(9),11]

**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,yes(3),5,6,no(8),yes(9),11,12,no(14),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.............

Selected alternative from the above list :
[5,6,yes(7),11]

**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,yes(3),5,6,yes(7),11,12,no(14),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.


.....**************.....
```

```
                    PROCESSING __ phase 3


..............

* Required pair : [5, 11] wrt var x, - c-use -.

Combine new alternative from source to 5
[1,2,no(4),5]

with a member of the known paths from 5 to 11 wrt var x :
[[5,6,no(8),no(10),11],[5,6,no(8),yes(9),11],[5,6,yes(7),11]]
''''''''''''''''

selected alternative from the first list :
[5,6,no(8),no(10),11]

and a member of the known paths from 11 to sink :
[[11,12,no(14),15,16,s],[11,12,yes(13),15,16,s]]
..............

selected alternative from the second list :
[11,12,no(14),15,16,s]


** Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,no(4),5,6,no(8),no(10),11,12,no(14),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[11,12,yes(13),15,16,s]


** Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,no(4),5,6,no(8),no(10),11,12,yes(13),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.

''''''''''''''''

selected alternative from the first list :
[5,6,no(8),yes(9),11]

and a member of the known paths from 11 to sink :
[[11,12,no(14),15,16,s],[11,12,yes(13),15,16,s]]
..............

selected alternative from the second list :
[11,12,no(14),15,16,s]


** Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,no(4),5,6,no(8),yes(9),11,12,no(14),15,16,s]
```

```
The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[11,12,yes(13),15,16,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,no(4),5,6,no(8),yes(9),11,12,yes(13),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the first list :
[5,6,yes(7),11]

and a member of the known paths from 11 to sink :
[[11,12,no(14),15,16,s],[11,12,yes(13),15,16,s]]
..............

selected alternative from the second list :
[11,12,no(14),15,16,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,no(4),5,6,yes(7),11,12,no(14),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.

..............

selected alternative from the second list :
[11,12,yes(13),15,16,s]


**  Candidate path to cover nodes from 5 to 11 wrt var x :
[1,2,no(4),5,6,yes(7),11,12,yes(13),15,16,s]

The candidate path is nonexecutable,
try another alternative, - if any.
..............@..............


* No  More  Paths *

**  Achieved  Coverage  :0.0000000000E+00%

**  Remaining  Pairs  :   5 --> 11,  wrt var  x .
```

```
                        Input  File
1  * source *
[read1([p,e]),(d gets 1),(x gets 0),(c gets(2 times p))]
Nonlocal definition of var :  p
Nonlocal definition of var :  e
Nonlocal definition of var :  d
Nonlocal definition of var :  x
Nonlocal definition of var :  c
[2]
.......__________.......
2
[(c less 2)]
P-use of var :  c
[yes(3),no(6)]
.......__________.......
3
[(d greater e)]
P-use of var :  d
P-use of var :  e
[yes(4),no(7)]
.......__________.......
4
[(d gets(d divreal 2)),(t gets((c minus(2 times x))plus d))]
Global c-use of var :  d
Nonlocal definition of var :  d
Global c-use of var :  c
Global c-use of var :  x
Nonlocal definition of var :  t
[5]
.......__________.......
5
[(t eqgreat 0)]
P-use of var :  t
[yes(9),no(8)]
.......__________.......
6
[]
[s]  * sink *
.......__________.......
7
[write1([x])]
Global c-use of var :  x
[s]  * sink *
.......__________.......
8
[(c gets(2 times c))]
Global c-use of var :  c
Nonlocal definition of var :  c
[3]
.......__________.......
9
[(x gets(x plus d)),(t gets(((2 times c)minus(2 times x))plus d))]
Global c-use of var :  x
Global c-use of var :  d
Nonlocal definition of var :  x
Global c-use of var :  c
Nonlocal definition of var :  t
[3]
.......__________.......
```

```
---------------
* Required pair : [9, 4] wrt var x, - c-use -.


Def-clear subpath from 9 to 4 wrt var x :
[9,3,yes(4)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [5,no(8),3,no(7),s]

**  Candidate path to cover nodes from 9 to 4 wrt var x :
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]

The candidate path is nonexecutable,
try another alternative, - if any.

---------------
* Required pair : [9, 7] wrt var x, - c-use -.


Def-clear subpath from 9 to 7 wrt var x :
[9,3,no(7)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [s]

**  Candidate path to cover nodes from 9 to 7 wrt var x :
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]

The candidate path is nonexecutable,
try another alternative, - if any.

---------------
* Required pair : [9, 9] wrt var x, - c-use -.


Def-clear subpath from 9 to 9 wrt var x :
[9,3,yes(4),5,yes(9)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [3,no(7),s]

**  Candidate path to cover nodes from 9 to 9 wrt var x :
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]

The candidate path is executable.
* Covered  pair : [9, 9] wrt var x, by the candidate path

---------------
* Required pair : [8, 4] wrt var c, - c-use -.
```

```
Def-clear subpath from 8 to 4 wrt var c :
[8,3,yes(4)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [5,no(8),3,no(7),s]

**  Candidate path to cover nodes from 8 to 4 wrt var c :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]

The candidate path is executable.
* Covered  pair : [8, 4] wrt var c, by the candidate path

---------------
* Required pair : [8, 8] wrt var c, - c-use -.


Def-clear subpath from 8 to 8 wrt var c :
[8,3,yes(4),5,no(8)]
* Covered  pair : [8, 8] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


---------------
* Required pair : [8, 9] wrt var c, - c-use -.


Def-clear subpath from 8 to 9 wrt var c :
[8,3,yes(4),5,yes(9)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [3,no(7),s]

**  Candidate path to cover nodes from 8 to 9 wrt var c :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]

The candidate path is executable.
* Covered  pair : [8, 9] wrt var c, by the candidate path

---------------
* Required pair : [4, 5] wrt var t, - p-use ---> branch yes -.

* Required pair : [4, 5] wrt var t, - p-use ---> branch no -.


Def-clear subpath from 4 to 5 wrt var t - yes branch - :
[4,5,yes(9)]
* Covered  pair : [4, 5] wrt var t, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


--------------

Def-clear subpath from 4 to 5 wrt var t - no branch - :
```

```
[4,5,no(8)]
* Covered  pair : [4, 5] wrt var t, by a known executable path- branch no -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


...............
* Required pair : [4, 4] wrt var d, - c-use -.


Def-clear subpath from 4 to 4 wrt var d :
[4,5,no(8),3,yes(4)]
* Covered  pair : [4, 4] wrt var d, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


...............
* Required pair : [4, 9] wrt var d, - c-use -.


Def-clear subpath from 4 to 9 wrt var d :
[4,5,yes(9)]
* Covered  pair : [4, 9] wrt var d, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


...............
* Required pair : [4, 3] wrt var d, - p-use ---> branch yes -.

* Required pair : [4, 3] wrt var d, - p-use ---> branch no -.


Def-clear subpath from 4 to 3 wrt var d - yes branch - :
[4,5,no(8),3,yes(4)]
* Covered  pair : [4, 3] wrt var d, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


..............

Def-clear subpath from 4 to 3 wrt var d - no branch - :
[4,5,no(8),3,no(7)]
* Covered  pair : [4, 3] wrt var d, by a known executable path- branch no -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
```

```
---------------
* Required pair : [1, 4] wrt var c, - c-use -.


Def-clear subpath from 1 to 4 wrt var c :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 4] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


---------------
* Required pair : [1, 8] wrt var c, - c-use -.


Def-clear subpath from 1 to 8 wrt var c :
[1,2,yes(3),yes(4),5,no(8)]
* Covered  pair : [1, 8] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


---------------
* Required pair : [1, 9] wrt var c, - c-use -.


Def-clear subpath from 1 to 9 wrt var c :
[1,2,yes(3),yes(4),5,yes(9)]
* Covered  pair : [1, 9] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


---------------
* Required pair : [1, 2] wrt var c, - p-use ---> branch yes -.

* Required pair : [1, 2] wrt var c, - p-use ---> branch no -.


Def-clear subpath from 1 to 2 wrt var c - yes branch - :
[1,2,yes(3)]
* Covered  pair : [1, 2] wrt var c, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


..............

Def-clear subpath from 1 to 2 wrt var c - no branch - :
[1,2,no(6)]

Expansions for the above subpath,
```

```
from source : []
to sink     : [s]

**  Candidate path to cover nodes from 1 to 2 wrt var c, - branch no -,  :
[1,2,no(6),s]

The candidate path is executable.
* Covered  pair : [1, 2] wrt var c, by the candidate path- branch no -

---------------
* Required pair : [1, 4] wrt var x, - c-use -.


Def-clear subpath from 1 to 4 wrt var x :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 4] wrt var x, by a known executable path
 from the list of found executable paths :
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


---------------
* Required pair : [1, 7] wrt var x, - c-use -.


Def-clear subpath from 1 to 7 wrt var x :
[1,2,yes(3),no(7)]

Expansions for the above subpath,
from source : []
to sink     : [s]

**  Candidate path to cover nodes from 1 to 7 wrt var x :
[1,2,yes(3),no(7),s]

The candidate path is nonexecutable,
try another alternative, - if any.

---------------
* Required pair : [1, 9] wrt var x, - c-use -.


Def-clear subpath from 1 to 9 wrt var x :
[1,2,yes(3),yes(4),5,yes(9)]
* Covered  pair : [1, 9] wrt var x, by a known executable path
 from the list of found executable paths :
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


---------------
* Required pair : [1, 4] wrt var d, - c-use -.


Def-clear subpath from 1 to 4 wrt var d :
[1,2,yes(3),yes(4)]
```

```
* Covered  pair : [1, 4] wrt var d, by a known executable path
 from the list of found executable paths :
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]

---------------
* Required pair : [1, 3] wrt var d, - p-use ---> branch yes -.

* Required pair : [1, 3] wrt var d, - p-use ---> branch no -.


Def-clear subpath from 1 to 3 wrt var d - yes branch - :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 3] wrt var d, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


..............

Def-clear subpath from 1 to 3 wrt var d - no branch - :
[1,2,yes(3),no(7)]

Expansions for the above subpath,
from source : []
to sink     : [s]

**  Candidate path to cover nodes from 1 to 3 wrt var d, - branch no -,  :
[1,2,yes(3),no(7),s]

The candidate path is a member of the nonexecutable paths :
[1,2,yes(3),no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [1, 3] wrt var e, - p-use ---> branch yes -.

* Required pair : [1, 3] wrt var e, - p-use ---> branch no -.


Def-clear subpath from 1 to 3 wrt var e - yes branch - :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 3] wrt var e, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]


..............
```

```
Def-clear subpath from 1 to 3 wrt var e - no branch - :
[1,2,yes(3),no(7)]

Expansions for the above subpath,
from source : [ ]
to sink     : [s]

**  Candidate path to cover nodes from 1 to 3 wrt var e, - branch no -,  :
[1,2,yes(3),no(7),s]

The candidate path is a member of the nonexecutable paths :
[1,2,yes(3),no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]

...............@...............



**  Achieved  Coverage  :0.8000000000E+02%

**  Remaining  Pairs  :   9 --> 4,  wrt var  x .
                          9 --> 7,  wrt var  x .
                          1 --> 7,  wrt var  x .
                          1 --> 3, - branch no -,  wrt var  d .
                          1 --> 3, - branch no -,  wrt var  e .
```

```
                          Input  File
1  * source *
[readl([p,e]),(d gets 1),(x gets 0),(c gets(2 times p))]
Nonlocal definition of var :  p
Nonlocal definition of var :  e
Nonlocal definition of var :  d
Nonlocal definition of var :  x
Nonlocal definition of var :  c
[2]
.......__________.......
2
[(c less 2)]
P-use of var :  c
[yes(3),no(6)]
.......__________.......
3
[(d greater e)]
P-use of var :  d
P-use of var :  e
[yes(4),no(7)]
.......__________.......
4
[(d gets(d divreal 2)),(t gets((c minus(2 times x))plus d))]
Global c-use of var :  d
Nonlocal definition of var :  d
Global c-use of var :  c
Global c-use of var :  x
Nonlocal definition of var :  t
[5]
.......__________.......
5
[(t eqgreat 0)]
P-use of var :  t
[yes(9),no(8)]
.......__________.......
6
[]
[s]  * sink *
.......__________.......
7
[writel([x])]
Global c-use of var :  x
[s]  * sink *
.......__________.......
8
[(c gets(2 times c))]
Global c-use of var :  c
Nonlocal definition of var :  c
[3]
.......__________.......
9
[(x gets(x plus d)),(t gets(((2 times c)minus(2 times x))plus d))]
Global c-use of var :  x
Global c-use of var :  d
Nonlocal definition of var :  x
Global c-use of var :  c
Nonlocal definition of var :  t
[3]
.......__________.......
```

```
                    PROCESSING -- phase 1


---------------
* Required pair : [9, 4] wrt var x, - c-use -.


Def-clear subpath from 9 to 4 wrt var x :
[9,3,yes(4)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [5,no(8),3,no(7),s]

**  Candidate path to cover nodes from 9 to 4 wrt var x :
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]

The candidate path is executable.
* Covered  pair : [9, 4] wrt var x, by the candidate path

---------------
* Required pair : [9, 7] wrt var x, - c-use -.


Def-clear subpath from 9 to 7 wrt var x :
[9,3,no(7)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [s]

**  Candidate path to cover nodes from 9 to 7 wrt var x :
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]

The candidate path is executable.
* Covered  pair : [9, 7] wrt var x, by the candidate path

---------------
* Required pair : [9, 9] wrt var x, - c-use -.


Def-clear subpath from 9 to 9 wrt var x :
[9,3,yes(4),5,yes(9)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [3,no(7),s]

**  Candidate path to cover nodes from 9 to 9 wrt var x :
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]

The candidate path is executable.
* Covered  pair : [9, 9] wrt var x, by the candidate path

---------------
* Required pair : [8, 4] wrt var c, - c-use -.
```

```
Def-clear subpath from 8 to 4 wrt var c :
[8,3,yes(4)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [5,no(8),3,no(7),s]

**  Candidate path to cover nodes from 8 to 4 wrt var c :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]

The candidate path is executable.
* Covered  pair : [8, 4] wrt var c, by the candidate path

---------------
* Required pair : [8, 8] wrt var c, - c-use -.


Def-clear subpath from 8 to 8 wrt var c :
[8,3,yes(4),5,no(8)]
* Covered  pair : [8, 8] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [8, 9] wrt var c, - c-use -.


Def-clear subpath from 8 to 9 wrt var c :
[8,3,yes(4),5,yes(9)]

Expansions for the above subpath,
from source : [1,2,yes(3),yes(4),5]
to sink     : [3,no(7),s]

**  Candidate path to cover nodes from 8 to 9 wrt var c :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]

The candidate path is executable.
* Covered  pair : [8, 9] wrt var c, by the candidate path

---------------
* Required pair : [4, 5] wrt var t, - p-use ---> branch yes -.

* Required pair : [4, 5] wrt var t, - p-use ---> branch no -.


Def-clear subpath from 4 to 5 wrt var t - yes branch - :
[4,5,yes(9)]
* Covered  pair : [4, 5] wrt var t, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]
```

```
t..............

Def-clear subpath from 4 to 5 wrt var t - no branch - :
[4,5,no(8)]
* Covered  pair : [4, 5] wrt var t, by a known executable path- branch no -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [4, 4] wrt var d, - c-use -.


Def-clear subpath from 4 to 4 wrt var d :
[4,5,no(8),3,yes(4)]
* Covered  pair : [4, 4] wrt var d, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [4, 9] wrt var d, - c-use -.


Def-clear subpath from 4 to 9 wrt var d :
[4,5,yes(9)]
* Covered  pair : [4, 9] wrt var d, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [4, 3] wrt var d, - p-use ---> branch yes -.

* Required pair : [4, 3] wrt var d, - p-use ---> branch no -.


Def-clear subpath from 4 to 3 wrt var d - yes branch - :
[4,5,no(8),3,yes(4)]
* Covered  pair : [4, 3] wrt var d, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]
```

```
.....

Def-clear subpath from 4 to 3 wrt var d - no branch - :
[4,5,no(8),3,no(7)]
* Covered  pair : [4, 3] wrt var d, by a known executable path- branch no -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [1, 4] wrt var c, - c-use -.


Def-clear subpath from 1 to 4 wrt var c :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 4] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [1, 8] wrt var c, - c-use -.


Def-clear subpath from 1 to 8 wrt var c :
[1,2,yes(3),yes(4),5,no(8)]
* Covered  pair : [1, 8] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [1, 9] wrt var c, - c-use -.


Def-clear subpath from 1 to 9 wrt var c :
[1,2,yes(3),yes(4),5,yes(9)]
* Covered  pair : [1, 9] wrt var c, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
```

```
* Required pair : [1, 2] wrt var c, - p-use ---> branch yes -.

* Required pair : [1, 2] wrt var c, - p-use ---> branch no -.


Def-clear subpath from 1 to 2 wrt var c - yes branch - :
[1,2,yes(3)]
* Covered  pair : [1, 2] wrt var c, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


...............

Def-clear subpath from 1 to 2 wrt var c - no branch - :
[1,2,no(6)]

Expansions for the above subpath,
from source : []
to sink     : [s]

**  Candidate path to cover nodes from 1 to 2 wrt var c, - branch no -,  :
[1,2,no(6),s]

The candidate path is executable.
* Covered  pair : [1, 2] wrt var c, by the candidate path- branch no -

---------------
* Required pair : [1, 4] wrt var x, - c-use -.


Def-clear subpath from 1 to 4 wrt var x :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 4] wrt var x, by a known executable path
 from the list of found executable paths :
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [1, 7] wrt var x, - c-use -.


Def-clear subpath from 1 to 7 wrt var x :
[1,2,yes(3),no(7)]

Expansions for the above subpath,
from source : []
to sink     : [s]

**  Candidate path to cover nodes from 1 to 7 wrt var x :
[1,2,yes(3),no(7),s]
```

```
The candidate path is executable.
* Covered  pair : [1, 7] wrt var x, by the candidate path

---------------
* Required pair : [1, 9] wrt var x, - c-use -.


Def-clear subpath from 1 to 9 wrt var x :
[1,2,yes(3),yes(4),5,yes(9)]
* Covered  pair : [1, 9] wrt var x, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),no(7),s]
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8,,3,no(7),s]


---------------
* Required pair : [1, 4] wrt var d, - c-use -.


Def-clear subpath from 1 to 4 wrt var d :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 4] wrt var d, by a known executable path
 from the list of found executable paths :
[1,2,yes(3),no(7),s]
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------
* Required pair : [1, 3] wrt var d, - p-use ---> branch yes -.

* Required pair : [1, 3] wrt var d, - p-use ---> branch no -.


Def-clear subpath from 1 to 3 wrt var d - yes branch - :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 3] wrt var d, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),no(7),s]
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------

Def-clear subpath from 1 to 3 wrt var d - no branch - :
```

```
[1,2,yes(3),no(7)]
* Covered  pair : [1, 3] wrt var d, by a known executable path- branch no -
 from the list of found executable paths :
[1,2,yes(3),no(7),s]
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


---------------

* Required pair : [1, 3] wrt var e, - p-use ---> branch yes -.

* Required pair : [1, 3] wrt var e, - p-use ---> branch no -.


Def-clear subpath from 1 to 3 wrt var e - yes branch - :
[1,2,yes(3),yes(4)]
* Covered  pair : [1, 3] wrt var e, by a known executable path- branch yes -
 from the list of found executable paths :
[1,2,yes(3),no(7),s]
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]


...............

Def-clear subpath from 1 to 3 wrt var e - no branch - :
[1,2,yes(3),no(7)]
* Covered  pair : [1, 3] wrt var e, by a known executable path- branch no -
 from the list of found executable paths :
[1,2,yes(3),no(7),s]
[1,2,no(6),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,no(8),3,yes(4),5,no(8),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,no(7),s]
[1,2,yes(3),yes(4),5,yes(9),3,yes(4),5,no(8),3,no(7),s]

...............@...............



** Achieved Coverage :0.1000000000E+03%

** Remaining Pairs :    No More Pairs
```

# CHAPTER VI

# BIBLIOGRAPHY

## 6.1 Cited References


[ 1] Adrion, Richards W., et al., "Validation, Verification, and Testing of Computer Software", ACM Computing Surveys, Vol. 14, No. 2, June 1982, pp. 159-192.

[ 2] Brooks, Frederick P., Jr., "No Silver Bullet: Essence and Accidents of Software Engineering", IEEE Computer, Vol. 20, No. 4, April 1987, pp. 10-19.

[ 3] Budd, Timothy A., "Mutation Analysis: Ideas, Examples, Problems and Prospects", in Computer Program Testing, by B. Chandrasekaran & S. Radicchi, SOGESTA 1981, pp. 129-148.

[ 4] Cavano, Joseph P., "Software Reliability Measurement: Prediction, Estimation, and Assessment", The Journal of Systems and Software, Vol. 4, No. 4, November 1984, pp. 269-275.

[ 5] Clarke, Lori A., et al., "A Close Look at Domain Testing", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, July 1982, pp. 380-390.

[ 6] Clarke, Lori A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976, pp. 215-222.

[ 7] Clarke, Lori A., et al., "Applications of Symbolic Evaluation", The Journal of Systems and Software, Vol. 5, No. 1, February 1985, pp. 15-35.

[ 8] Coward, David P., "Determining Path Feasibility for Commercial Programs", SIGPLAN Notices, Vol. 23, No. 3, March 1988, pp. 93-101.

[ 9] Davis, Martin, "Hilbert's Tenth Problem Is Unsolvable", The American Mathematical Monthly, Vol. 80, No. 3, March 1973, pp. 233-269.

[10] DeMillo, Richard A., et al., "Hints on Test Data Selection: Help for the Practicing Programmer", IEEE Computer, Vol. 11, No. 4, April 1978, pp. 34-41.

[11] Frankl, Phyllis G., et al., "ASSET: A System to Select and Evaluate Tests", Proceedings of the IEEE Conference on Software Tools, New York, April 1985, pp.72-79.

[12] Frenkel, Karen A., "Toward Automating the Software Development Cycle", Communications of the ACM, Vol. 28, No. 6, June 1985, pp. 578-589.

[13] Gaffney, John E., Jr., "Estimating the Number of Faults in Code", IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, July 1984, pp. 459- 464.

[14] Goel, Amrit L., "Software Reliability Models: Assumptions, Limitations, and Applicability", IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 1411-1423.

[15] Goodenough, John B., Gerhart, Susan L., "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 156-173.

[16] Howden, William E., "Methodology for the Generation of Program Test Data", IEEE Transactions on Computers, Vol. C-24, No. 5, May 1975, pp. 554-559.

[17] Howden, William E., "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976, pp. 208-215.

[18] Ince, D. C., "The Automatic Generation of Test Data", The Computer Journal, Vol. 30, No. 1, February 1987, pp. 63-69.

[19] Joshi, Ramchandra D., "Software Development for Reliable Software Systems", The Journal of Systems and Software, Vol. 3, No. 2, June 1983, pp. 107-121.

[20] Kobler, Virginia P., McDaniel, Bonnie G., "Expert Systems: Status and Perspectives", IEEE Proceedings of Computer Software & Applications Conference, November 8-12, 1982, pp. 565-570.

[21] Laski, Janusz W., Korel, Bogdan, "A Data Flow Oriented Program Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 347-354.

[22] Lipow, M., "Number of Faults per Line of Code", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, July 1982, pp. 437-439.

[23] Miller, Edward F., Jr., "Program Testing: Art Meet Theory", IEEE Computer, Vol. 10, No. 7, July 1977, pp. 42-51.

[24] Myers, Glenford J., The Art of Software Testing, John Wiley & Sons, Inc., 1979.

[25] Ntafos, Simeon C., "On Required Element Testing", IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, November 1984, pp. 795-803.

[26] Ntafos, Simeon C., "A Comparison of Some Structural Testing Strategies", IEEE Transactions on Software Engineering, Vol. 14, No. 6, June 1988, pp. 868-874.

[27] Prather, Ronald E., Myers, J. Paul, Jr., "The Path Prefix Software Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987, pp. 761-766.

[28] Ramamoorthy, C. V., et al., "On the Automated Generation of Program Test Data", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976 pp. 293-300.

[29] Rapps, Sandra, Weyuker, Elaine J., "Selecting Software Test Data Using Data Flow Information", IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April 1985, pp. 367-375.

[30] Rubin, Howard A., et al., "Integrating Software Development Estimation, Planning, Scheduling, and Tracking: The PLANMACS System", IEEE Proceedings of SOFTAIR, Conference II, December 1985.

[31] Schick, George J., Wolverton, Ray W., "An Analysis of Competing Software Reliability Models", IEEE Transactions on Software Engineering, Vol. SE-4, No. 2, March 1978, pp. 104-120.

[32] Schneider, Victor, "Approximations for the Halstead Software Science Software Error Rate and Project Effort Estimations", SIGPLAN Notices, Vol. 23, No. 1, January 1988, pp. 40-47.

[33] Simon, Herbert A., "Whether Software Engineering Needs to be Artificially Intelligent", IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986, pp. 726-732.

[34] Stetter, Franz, "Comments on "Number of Faults per Line of Code"", IEEE Transactions on Software Engineering, Vol. SE-12, No. 12, December 1986, pp. 1145.

[35] Thayer, Thomas A., et al., Software Reliability, TRW Series of Software Technology 2, North - Holland Publishing Company, 1978.

[36] Voges, Udo, et al., "SADAT - An Automated Testing Tool", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 286-290.

[37] Weiser, M. D., "Comparison of Structural Test Coverage Metrics", IEEE SOFTWARE, Vol. 1, No. 2, March 1985, pp. 80-85.

[38] Weyuker, Elaine J., Ostrand, Thomas J., "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 236-246.

[39] White, Lee J., "Basic Mathematical Definitions and Results in Testing", in Computer Program Testing, by B. Chandrasekaran & S. Radicchi, SOGESTA, 1981, pp. 13-24.

[40] White, Lee J., Cohen, Edward I., "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 247-257.

## 6.2 Uncited References


[ 1] Albrecht, Allan J., Gaffney, John E., Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November 1983, pp. 639-648.

[ 2] Andrews, Dorothy M., Benson, Jeoffrey P., "An Automated Program Testing Methodology and its Implementation", Proceedings, Fifth International Conference on Software Engineering, 1981, pp. 254-261.

[ 3] Balzer, Robert, et al., "Software Technology in the 1990's: Using a New Paradigm", IEEE Computer, Vol. 16, No. 11, November 1983, pp. 39-45.

[ 4] Basili, Victor R., Perricone, Barry T., "Software Errors and Complexity: An Empirical Investigation" Communications of the ACM, Vol. 27, No. 1, January 1984, pp. 42-52.

[ 5] Basili, Victor R., Selby, R. W., "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, December 1987, pp. 1278-1296.

[ 6] Boehm, Barry W., Standish, Thomas A., "Software Technology in the 1990's: Using an Evolutionary Paradigm", IEEE Computer, Vol. 16, No. 11, November 1983, pp. 30-37.

[ 7] Boehm, Barry W., "Improving Software Productivity", IEEE Computer, Vol. 20, No. 9, September 1987, pp. 43-57.

[ 8] Bouge, L., et al., "Test Cases Generation from Algebraic Specifications Using Logic Programming", The Journal of Systems and Software, Vol. 6, No. 4, November 1986, pp. 343-360.

[ 9] Chapman, David, "A Program Testing Assistant", Communications of the ACM, Vol. 25, No. 9, September 1982 pp. 625-634.

[10] Chow, T. S., "Testing Software Design Modeled by Finite State Machines", IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, May 1978, pp. 178-187.

[11] Christensen, K., et al., "A Perspective on Software Science", IBM Systems Journal, Vol. 20, No. 4, 1981, pp. 372-387.

[12] Chusho, Takeshi, "Test Data Selection and Quality Estimation Based on the Concept of Essential Branches for Path Testing", IEEE Transactions on Software Engineering, Vol. SE-13, No. 5, May 1987, pp. 509-517.

[13] Duke, M. O., "Testing in a Complex Systems Environment", IBM Systems Journal, Vol. 14, No. 4, 1975, pp. 353-365.

[14] Dunham, Janet R., Kruesi, Elizabeth, "The Measurement Task Area", IEEE Computer, Vol. 16, No. 11, November 1983, pp. 47-54.

[15] Dunn, Robert H., Software Defect Removal, McGraw Hill Book Company, 1984.

[16] Fagan, Michael E., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 182-211.

[17] Fagan, Michael E., "Advances in Software Inspections", IEEE Transactions on Software Engineering, Vol. SE-12, No. 7, July 1986, pp. 744-751.

[18] Fitzsimmons, Ann, Love, Tom, "A Review and Evaluation of Software Science", ACM Computing Surveys, Vol. 10, No. 1, March 1978, pp. 3-18.

[19] Foster, Kenneth A., "Error Sensitive Test Cases Analysis", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 258-264.

[20] Gabow, Harold N., et al., "On Two Problems in the Generation of Program Test Paths", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, September 1976, pp. 227-231.

[21] Hayes-Roth, Frederick, "The Knowledge-Based Expert System: A Tutorial", IEEE Computer, Vol. 17, No. 9, September 1984, pp. 11-28.

[22] Hayes-Roth, Frederick, et al., Building Expert Systems, Addison-Wesley Publishing Company Inc., 1983.

[23] Howden, William E., "The Theory and Practice of Functional Testing", IEEE SOFTWARE, Vol. 1, No. 5, September 1985, pp. 6-17.

[24] Howden, William E., "A Functional Approach to Program Testing and Analysis", IEEE Transactions on Software Engineering, Vol. SE-12, No. 10, October 1986, pp. 997-1005.

[25] Howden, William E., "Functional Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 2, March 1980, pp. 162-169.

[26] Howden, William E., "A Survey of Dynamic Analysis Methods", IEEE Tutorial: Software Testing and Validation Techniques, by E. Miller & W. Howden, 1981, pp. 209-231.

[27] Howden, William E., "Functional Testing and Design Abstractions", The Journal of Systems and Software, Vol. 1, 1980, pp. 307-313.

[28] Howden, William E., "Applicability to Software Validation Techniques to Scientific Programs", ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, July 1980, pp. 307-320.

[29] Howden, William E., "Errors, Design Properties and Functional Program Tests", in Computer Program Testing, by B. Chandrasekaran & S. Radicchi, SOGESTA, 1981, pp. 115-127.

[30] Howden, William E., "Validation of Scientific Programs", ACM Computing Surveys, Vol. 14, No. 2, June 1982, pp. 193-227.

[31] Howden, William E., "Life-Cycle Software Validation", IEEE Computer, Vol. 15, No. 2, February 1982, pp. 71-78.

[32] Howden, William E., "Theoretical and Empirical Studies of Program Testing", IEEE Transactions on Software Engineering, Vol. SE-4, No. 4, July 1978, pp. 293-298.

[33] Howden, William E., "An Evaluation of the Effectiveness of Symbolic Testing", Software-Practice & Experience, Vol. 8, No. 4, July-August 1978, pp. 381-397.

[34] Huang, J. C., "An Approach to Program Testing", ACM Computing Surveys, Vol. 7, No. 3, September 1975, pp. 113-128.

[35] Huang, J. C., "Detection of Data Flow Anomaly through Program Instrumentation", IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979, pp. 226-236.

[36] Iannino, Anthony, et al., "Criteria for Software Reliability Model Comparisons", IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, November 1984, pp. 687-691.

[37] Jones, Capers T., "Measuring Programming Quality and Productivity", IBM Systems Journal, Vol. 17, No. 1, 1978, pp. 39-63.

[38] Jones, Carole L., "A Process-Integrated Approach to Defect Prevention", IBM Systems Journal, Vol. 24, No. 2, 1985, pp. 150-167.

[39] Littlewood, Beverly, "Teories of Software Reliability: How Good Are They and How Can They Be Improved", IEEE Transactions on Software Engineering, Vol. SE-6, No. 5, September 1980, pp. 489-500.

[40] McCabe, Thomas J., "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976, pp. 308-320.

[41] McCabe, Thomas J., "Structured Testing: A Testing Methodology using the McCabe Complexity Metric", in Structured Testing by Thomas J. McCabe, 1983, IEEE Computer Society Press, pp. 19-47.

[42] Majoros, Maria, Sneed, Harry M., "The SOFTEST Program Test System", The Journal of Systems and Software, Vol. 2, No. 4, December 1981, pp. 289-296.

[43] Miller, Edward F., Jr., Melton, R. A., "Automated Generation of Testcase Datasets", Proceedings, International Conference on Software Engineering, 1975, pp. 51-58.

[44] Miller, Edward F., Jr., "Experience with Industrial Software Quality Testing", in Computer Program Testing, by B. Chandrasekaran & S. Radicchi, SOGESTA, 1981, pp. 265-277.

[45] Misra, P. N., "Software Reliability Analysis", IBM Systems Journal, Vol. 22, No. 3, 1983, pp. 262-270.

[46] Mohanty, Siba N., "Models and Measurements for Quality Assessment of Software", ACM Computing Surveys, Vol. 11, No. 3, September 1979, pp. 251-275.

[47] Paige, Michael R., "On Partitioning Program Graphs" IEEE Transactions on Software Engineering, Vol. SE-3, No. 6, November 1977, pp. 386-393.

[48] Prather, Ronald E., "An Axiomatic Theory of Software Complexity Measure", The Computer Journal, Vol. 27, No. 4, November 1984, pp. 340-347.

[49] Ramamoorthy, C. V., Bastani, Farokh B., "Software Reliability - Status and Perspectives", IEEE Transactions on Software Engineering, Vol. SE-8, No. 4, July 1982, pp. 354-371.

[50] Richardson, Debra J., Clarke, Lori A., "Partition Analysis: A Method Combining Testing and Verification", IEEE Transactions on Software Engineering, Vol. SE-11, No. 12, December 1985, pp. 1477-1490.

[51] Richardson, Debra J., Clarke, Lori A., "A Partition Analysis Method to Increase Program Reliability", Proceedings, Fifth International Conference on Software Engineering, 1981, pp. 244-253.

[52] Tai, Kuo-Chung, "A Program Complexity Metric Based on Data Flow Information in Control Graphs", 7th International Conference on Software Engineering, March 26-29, 1984, Orlando, Florida, pp. 239-248.

[53] Tai, Kuo - Chung, "Program Testing Complexity and Test Criteria", IEEE Transactions on Software Engineering, Vol. SE-6, No. 6, November 1980, pp. 531-538.

[54] Velasco, F. R. D., "A Method for Test Data Selection", The Journal of Systems and Software, Vol. 7, No. 2, June 1987, pp. 89-97.

[55] Weyuker, Elaine J., "Axiomatizing Software Test Data Adequacy", IEEE Transactions on Software Engineering, Vol. SE-12, No. 12, December 1986, pp. 1128-1138.

[56] Weyuker, Elaine J., "The Complexity of Data Flow Criteria for Test Data Selection", Information Processing Letters, Vol. 19, No. 2, August 1984, pp. 103-109.

[57] Woodward, Martin R., et al., "Experience with Path Analysis and Testing of Programs", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, May 1980, pp. 278-286.

[58] Woodward, Martin R., et al., "A Measure of Control Flow Complexity in Program Text", IEEE Transactions on Software Engineering, Vol. SE-5, No. 1, January 1979, pp. 45-50.

[59] Zeil, Steven J., "Testing for Perturbations of Program Statements", IEEE Transactions on Software Engineering, Vol. SE-9, No. 3, May 1983, pp. 335-346.

[60] Zualkernan, I., et al., "Expert Systems and Software Engineering: Ready for Marriage", IEEE EXPERT, Winter 1986, pp. 24-31.