CANADIAN THESES

THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc.) ne sont pas microfilmés.

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30.

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

SOFTWARE TEST DESIGN BASED ON PATH SENSITIZATION

by

Sophocles Arabatzis

Thesis

submitted to the School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Masters

of

Computer Science

Department of Computer Science

University of Ottawa

October 1986

ISBN 0-315-36462-9

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

The University of Ottawa requires the signatures of all persons using or photocopying this document. Please sign below, and give address and date.

ACKNOWLEDGEMENTS

I wish to express my deep gratitude and sincere appreciation to my thesis supervisor, Professor Dr. Robert L. Probert, for his outstanding advice, guidance, patience, and many useful suggestions during the research and preparation of this thesis. His generous financial support provided throughout the duration of my graduate work is gratefully acknowledged.

I am highly indebted and grateful to my wife, Nicky, for the enormous patience, constant encouragement, and loving support she showed during my studies.

I am thankful too for the financial support I got during the early stages of my graduate work from Professor Dr. To-Yat Cheung, from the University of Ottawa for the Graduate Research Scholarship, and from Simware Inc. for the Simware Award.

Finally, I would like to extend my thanks to many friends, colleagues, and all the staff of the Department of Computer Science, University of Ottawa, for their support and assistance.

To my wife

ABSTRACT

Often, validation of functional requirements has been carried out in an informal and unsystematic way. This thesis propposes a systematic approach to validation. The method involves constructing tests that are based on the important features of both requirements and design.

This approach is based on the use of a combination of logic-based testing techniques such as path sensitization and cause-effect graphing. We develop an incremental sensitized test case design methodology to systematically guide the selection of non-redundant high-yield test cases. We demonstrate the feasibility of our approach by designing and implementing a prototype support tool implementation, called DINSERT. DINSERT is applied to a standard "benchmark" problem for testing techniques, the "Text Reformatter Problem". The results compare favorably to other test design approaches. In our experience, DINSERT provides an efficient and effective environment for requirements-based testing tool.

This thesis focuses on the functional testing of computer software. However, the methodology described here is equally applicable to firmware and hardware. Our approach can be applied to early stages in the software development process, and appears to have considerable potential for detecting ambiguities, inconsistencies, and omissions.

We believe that our incremental sensitized test design method is effective and efficient, and it will help to produce more reliable specifications, more effective tests, and thus, more reliable software.

ABBREVIATIONS

CCF	:	*Cause-Effect Combinational Function*
CCN	:	*Cause-Effect Combinational Network*
CEG	:	*Cause-Effect Graphing*
CF	:	*Combinational Function*
CN	:	*Combinational Network*
DINSERT	:	*INcremental SEnsitized Test Requirement Design*
ENF	:	*Equivalent Normal Form*
GTF	:	*Gate Traversal Form*
PoS	:	*Product of Sum-terms Canonical Form*
SoP	:	*Sum of Product-terms Canonical Form*
TRP	:	*Text Reformatter Problem*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter I
INTRODUCTION AND BACKGROUND

1.1 An Overview of Software Testing

Software engineering is the design and analysis of techniques and tools for producing software systems which are cost effective and reliable. A fundamental requirement in software engineering is the need to interpret and apply sound engineering discipline and practice to the *design*, *development*, *testing*, and *maintenance* of software systems [ViRa 84].

Program testing is the symbolic or physical execution of a set of test cases with the intent of exposing embedded faults in the program. Testing of software is of primary concern to software engineers. In fact good software testing is as important and as intellectually challenging an activity as software design [Dunn 84].

1.1.1 Foundations of Software Testing

Software testing is a rapidly maturing area within software engineering that is receiving increased attention both by computer science theoreticians and practitioners. Its general aim is to assess the quality of software systems under carefully controlled circumstances.

Hardware testing has developed as an engineering discipline. Many of the techniques in hardware testing are applicable or related to software testing. Path sensitization testing techniques for hardware inspired parts of our approach. These techniques are surveyed in chapter II, and include the truth table method, path sensitization applied to both single paths and combinations of paths, the D-algorithm, equivalent-normal form, and diagnosing trees. But testing software requires additional techniques as well.

In June 1972, the University of North Carolina hosted the first conference devoted to *software testing*. Since then there have been seven International Software Engineering Conferences, and a spate of smaller workshops and symposia devoted either fully or in part to software testing. Yet the history of testing goes back to the beginnings of computing. One of Turing's early papers stated that "*testing is the empirical form of software quality assurance, while proving is the theoretical way*".

As Turing had indicated, there are two approaches to software verification: *Program proving* and *program testing*. Program proving is more formal and mathematical while program testing is more practical and heuristic [Goel 85]. The standard *program proving* approach is known as the *inductive assertion method*. This method has been investigated by Floyd, Hoare, Dijkstra, and recently by Reynolds [Reyn 81]. Program proving is not adequate for verifying program correctness. Garhart and Yelowitz [GaYe 76], showed several programs which were proved to be

correct, but was found that there were still containing faults. The faults were largely due to the difficulty of defining what exactly to prove.

Like program proving, program testing does not provide a guarantee of program correctness. A given testing strategy may be good for exposing certain kinds of faults in the program, but not for all possible kinds of faults [Myer 79]. An advantage of testing is that it can provide useful information about a program's actual behavior in its intended computing environment.

In practice neither proving nor testing can guarantee complete confidence in the correctness of a program. Each have its advantages and limitations and should be viewed as complementary methods for decreasing the likelihood of program failure. However, software testing is much more widely accepted as an intrinsic part of software engineering.

In 1975, Goodenough and Gerhart [GoGe 75] published a paper that tried to provide a theoretical foundation for testing. This paper presents a "*fundamental theorem of testing*" which characterizes a completely effective test selection strategy. A test selection strategy is completely effective if it is guaranteed to discover any error in a program. No such strategy has been found; however, some strategies have been developed which are guaranteed to discover particular types of errors [Howd 81]. On the other hand it is known that no uniform mechanical procedure can exist for finding a complete test set [Howd 76]. In many cases, however, this is not a serious practical limitation.

1.1.2 The Testing Process

Software testing requires both a rigorous plan and the consistent application of it to all parts of a software system. A systematic methodology is essential if the testing is going to be *both* thorough and economical.

The testing process consists of *pre-release testing* and *regression testing*. The pre-release testing activity is generally divided into *module* or *unit testing* and *system testing* [Turn 84]. Module testing is the verification[1] of a single program module in an isolated environment (i.e., isolated from all other modules). Module verification may also include mathematical proofs. The goal is to verify correctness of the design and coding with respect to the module specification. After module testing has been performed, *integration, sanity* or *interface testing* is carried out. The system is then tested against its functional specifications (*function testing*), its objectives (*system testing*), and its requirements (*acceptance testing*). Finally, *installation testing* and *performance testing* activities are carried out.

Briefly, *system testing* consists of validation[2] of the system with respect to its initial objectives. System testing is performed in a real environment. With regard to *performance testing*, many systems have specific performance or efficiency objectives, stating such properties as response times and

---

[1] *Verification* is an attempt to find errors by executing a program in a test or simulated environment [Myer 76].

[2] *Validation* is an attempt to find errors by executing or interpreting a program in a given real environment [Myer 76].

throughput rates under certain workload and configuration conditions. Thus, test cases must be designed that attempt to show that the system does not satisfy its performance objectives.

The purpose of the *function test* is to find any harmful discrepancies between the system and its external specification. The prerequisite for a successful function test is a precise and accurate external specification.

Regression testing, on the other hand, is normally performed after making a functional improvement or repair to the system. Its purpose is to determine if other aspects of the system have regressed because of the change.

Software testing encompasses a range of activities that are quite similar to the sequence of software development. These activities include:

- Establishment of test objectives;
- Test case design;
- Implementation of a suite of test cases;
- Development of a test harness (i.e., driver, stubs);
- Preparation of the SUT (System Under Test);
- Test execution and report generation;
- Evaluation of test results; and
- Evaluation of quality of test

Of these activities, the *test case design* is the most crucial. In this thesis, we focus on test requirement design. We briefly review some background material in the next section.

1.1.3 Test Case Design

The most important step in program testing is the design of *effective test cases*. The importance of test case design stems from the fact that "complete" or "exhaustive" testing is impractical. A test of any program must be necessarily incomplete (i.e., non-exhaustive testing cannot guarantee that there are no errors). Thus, given constraints on *personnel, time, budget, computer time*, etc., the key issue of testing becomes:

> *What subset of all possible test cases is most likely to detect serious errors?*

The study of test case design methodologies supplies us with partial answers to this question [Myer 79]. Probably the poorest method of all is random-input testing. This is testing a program by selecting at random some subset of all possible input values. A randomly selected collection of test cases has a low probability of detecting serious errors, and is therefore not a good approach to test case design [Myer 76,79, Mill 81].

Software test design techniques are usually classified as *Black-Box Testing* or *White-Box Testing* techniques. For computer software, black-box testing alludes to tests that are constructed at the user interface without knowledge of the internal structure. That is, test cases are intended to demonstrate that specified software functions are implemented, that input is properly accepted and output is correctly produced, and that the integrity of data files is maintained. If a formal specification of the system's input/output relationship is available a thorough set of black-box tests can be generated systematically [PrUr 84].

1.1.3.1 White-Box Tools and Techniques

White-box testing techniques come from many disciplines: *graph theory*, *programming languages*, and *reliability theory*. As an example, graphs are used to model the control flow and data flow in a program [Prob 82a]. In control flow graphs, *arcs* correspond to *branches* in the *program* and *nodes* correspond to *straight-line* segments in a *program*. Data flow graphs are used to examine the data flow between various program segments, and to design test cases to exercise all paths of interest [Howd 76]. A variety of tools exists, mainly in prototype form, to support various white-box testing techniques. These include:

- *Symbolic execution/evaluation systems*, which interactively simulate the execution of source text. These tools provide synoptic expressions of what the code under test will generate. In general, these tools can reveal defects to the extent that the specifications can be expressed with mathematical rigor [Boye 75, King 76, Clar 76, Howd 77, Chea 79].
- *Assertion based systems*, which process ancillary information about expected program behavior stated as run-time assertions about the current program state [StFo 75, AnBe 81]. Executing assertions generally requires the co-operation of the compiler [Wort 79]. An example is the use of range specifications for variables supported by some PASCAL compilers [Ande 79, Muss 80].

- *Data flow analyzers*, which perform a static analysis of the program searching for the presence of potential errors, called *data flow anomalies* [OsFo 76, FoOs 76, Howd 76, TaOs 80, Prob 82a].
- *Code coverage measurement systems*, which are generally based on the percentage of statements executed, branches selected, or paths traversed. The first is the weakest measure of test adequacy. The last is the strongest, however, 100% path coverage is practically unattainable [Howd 75, Mill 77,78, Prob 82a,82b].
- *Program mutation*, which is a way of measuring the effectiveness of test cases for finding defects. Simple alterations of the program called *mutants* are made to determine if the test cases can evoke different responses. Mutation testing requires mechanization of the mutation process to generate a sufficiently large number of mutants that the results are statistically significant [Budd 78,80, DeMI 78, Howd 82].

To give an example of one of the above white-box test design strategies, we consider the code coverage-directed approach in more detail. Exercising all possible outcomes of every decision statement at least once is considered a minimally thorough coverage criterion that is possible to attain in practice.

However, exhaustive white-box testing is impractical. For even small programs, the number of potentially executable logical paths can be very large [Myer 76,79, Pets 85, Pres 82, Dunn 84].

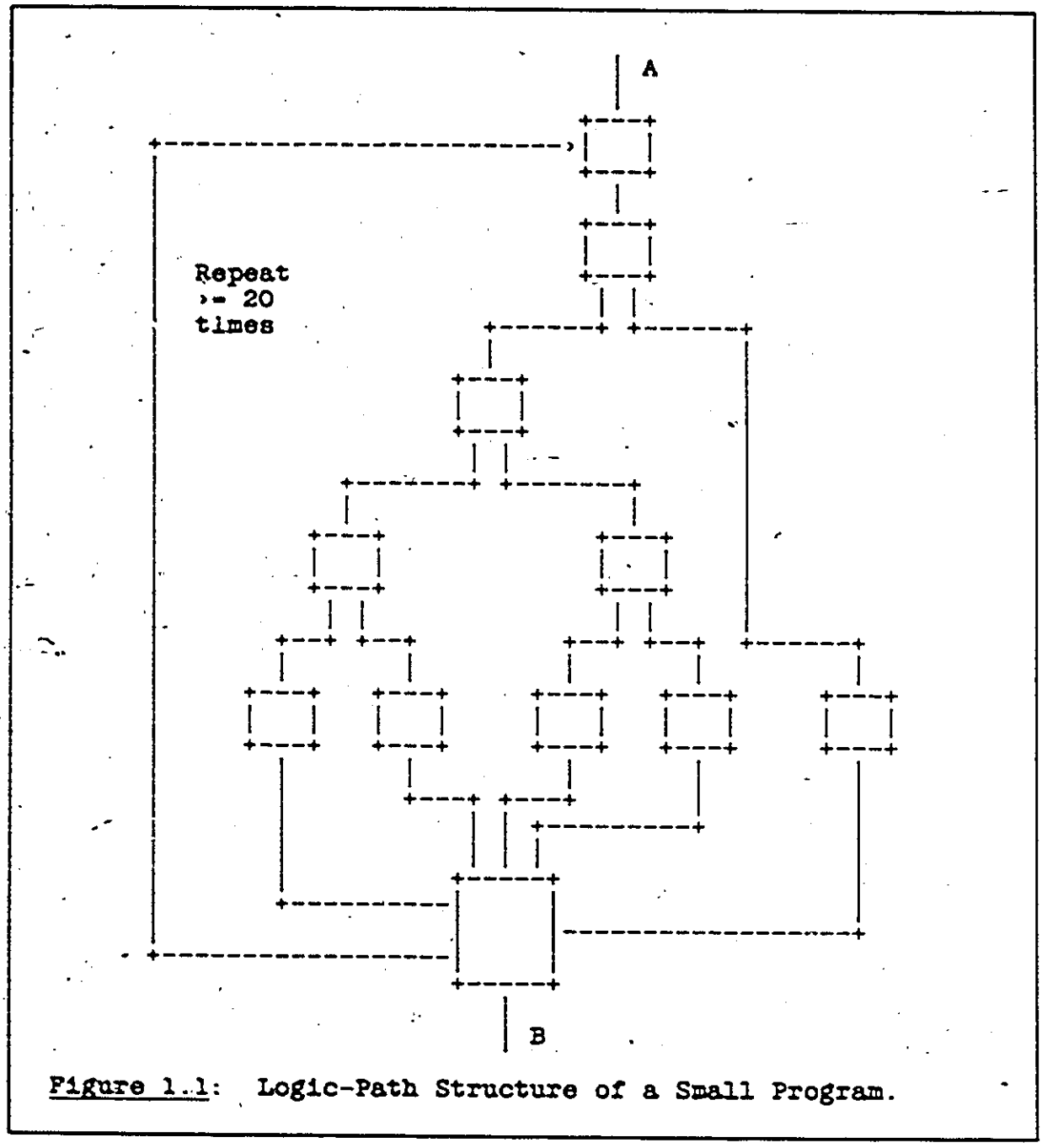


Figure 1.1: Logic-Path Structure of a Small Program.

For example, the number of distinct paths in a program with the structure given in Figure 1.1, is approximately 100-trillion! This assumes that all decision predicates are mutually independent.

In actual programs every decision is not independent from every other decision [Myer 79]. Thus, most potential execution paths are unfeasible paths, impossible to execute. As a result, path coverage is an unrealistic, unattainable coverage criterion. White-box test design strategies will not necessarily detect errors caused by missing logic, nor errors caused by incorrect predicate calculations. Suppose, for example, in a program, a coverage test may be erroneous expressed as:

IF ((A-B) < EPSILON) ...

This statement contains an error, namely (A-B), instead of taking the absolute value i.e., ($|A-B|$). Detection of this error, however, is dependent upon the test values used for A and B during testing and would not necessarily be detected.

White-box testing should not, however, be dismissed as impractical. Both black-box and white-box testing should be combined. This results in the following recommended [Myer 79] testing strategy:

1. Use certain effective black-box based test case design methodologies;
2. Monitor logic coverage achieved; and
3. Add new test cases to achieve a desired level of logic coverage.

1.1.3.2 Black-Box Techniques and Cause-Effect Graphing

The other class of strategies is called *black-box testing*, in which the tester views the program as a *black-box*; i.e., the tester is unaware of the internal behavior and structure of the program. The tester is only interested in the externally observable behavior of the program.

Four common *black-box* test design techniques are *Equivalence Partitioning*, *Boundary-Value Analysis*, *Cause-Effect Graphing*, and *Error Guessing*. The interested reader should refer to [Myer 79] for more information. The approach we use is inspired by cause-effect graphing, briefly described next and illustrated in more detail in Section 2.2.1.

One weakness of boundary-value analysis and equivalence partitioning is that neither explores *combinations* of input circumstances [Myer 79]. The testing of all combinations of input conditions is not a simple test, because the number of combinations is usually very large. If you had no systematic way of selecting a subset of combinations of input conditions, an arbitrary subset is usually selected. This would lead to an inefficient test or an ineffective test or both.

Cause-Effect Graphing (abbreviated CEG) is a technique that Elmendorf adopted from hardware testing in 1973 [Elme 73]. Since then, researchers have come up with several papers and software tools that either describe or further investigate this strategy or do both [Elme 74,75, Myer 76,79, Howd 80, AdGr 83, Wals 83, Prob 85].

CEG is a technique that aids in selecting, in a systematic way, a *high-yield* set of test cases. It also has a beneficial side effect in pointing out *incompleteness* and *ambiguities* in the *specifications* [Myer 79]. The cause-effect graph is actually a *combinatorial logic network*. However, rather than using standard electronics notation, a somewhat simpler notation is used. No knowledge of electronics is necessary other than an understanding of *Boolean logic* (i.e., understanding the *logical operators*: IDENTITY, AND, OR, and NOT).

CEG requires the representation of causal relationships in a system specification as a Boolean logic network. CEG is discussed in more detail in Section 2.2.1, while the graphical representation of the *logical operators* and the basic cause-effect graph symbols are shown in Figure 2.11 on page 53. The outcome of CEG is a set of requirements to be met by test data. CEG design consists of the following steps:

1. Represent the natural-language specification by a CEG;
2. Derive test requirements;
3. Represent test requirements as a decision table; and
4. Solve for test data.

An apparently difficult aspect of the technique is the conversion of a set of test requirements into a decision table; however, this step is algorithmic, and can be automated. A more difficult aspect is the derivation of actual test data from the set of test requirements. We expand on this point later in Section 5.4.

1.1.3.3 Complementary Test Design Strategy

A number of authors including Myers advocate a complementary test design strategy which is essentially the following:

1. If the specification contains combinations of input conditions, start with *cause-effect graphing*.
2. In any event, use *boundary-value analysis* [Myer 76,79], remember that this is an analysis of *input* and *output* boundaries. The boundary-value analysis yields a set of supplemental test conditions, but, as noted in the section on Cause-Effect Graphing, many or all of these can be incorporated into the cause-effect tests.
3. Identify the *valid* and *invalid equivalence classes* for the input and output, supplement the test cases identified above if necessary.
4. Use the *error-guessing* technique [Myer 76,79], to add additional test cases.
5. Examine the program's logic with regard to the set of test cases. Use either mutation testing or some coverage criterion that has not been met by the test cases identified in the prior four steps, and if meeting the criterion is not possible (i.e., certain combinations of conditions may be impossible to create because of the nature of the program), then, add sufficient test cases to cause the criterion to be satisfied.

1.2 Thesis Objectives and Approach

Our primary objectives in this thesis consists of the following:

1. To formulate a new, effective black-box test case design approach;
2. To demonstrate its effectiveness;
3. To design an effective tool to support this approach; and
4. To demonstrate the quality of the design of this tool.

This black-box test requirement design approach is applicable to high-level testing [PrUr 84]. In this respect, our work agrees with recent investigations [Fost 80] suggesting that testing activities should be carried out much earlier, in particular, at the specification [PrUr 82] and design phases [Prob 82b].

To obtain our new approach we melt effective *hardware* and *software* approaches, namely the Equivalent Normal Form (abbreviated ENF) and Cause-Effect Graphing (abbreviated CEG) techniques, respectively. The major steps in the research plan were:

1. To formalize our approach and apply it to a standard benchmark for testing techniques, the well-known Text Reformatter Problem (abbreviated TRP);
2. To compare it to two competing approaches (ENF and CEG);
3. To design and implement a prototype tool (called DINSERT), to assist in evaluating the approach; and
4. To use DINSERT on the benchmark problem (and other problems) to evaluate the completeness and effectiveness of its design.

Briefly, our approach to test case design problems involves two steps; the first is manual; the second is automated:

1. *Manual step*: Given a natural-language specification we use our *standard CCN* to represent a functional specification as a set of logical relationships between "*causes*" and "*effects*". A *standard CCN* is a formal language providing a rigorous way of representing a natural language specification by a more formal specification. This is a quite difficult step and comprises the starting point in our approach.
2. *Automated step*: Given a *standard CCN*, we proceed with its transformation into *standard CCF*, involving a ternary logical domain. This is the first automated step in our approach followed by other automated steps leading to a design of non-redundant high-yield test requirements.

We believe that our methodology can quickly be adapted to software functional testing. It assists the disciplined design of specification based black-box test cases. It is useful, effective, and has considerable potential to help testers save time and effort in designing, coding and testing or proving inappropriate programs. In particular, program proving is wasteful if the underlying specifications are invalid, inconsistent and inappropriate. On the other hand, our prototype system is, as far as we know, the first computerized tool of its kind automating the test case design process in a user-friendly and attractive interactive environment. While this thesis deals

with the functional testing of computer software, the methodology described here is equally applicable to *firmware* and *hardware*.

1.3 Motivation for Path Sensitization

Our investigation of one class of black-box test case design techniques called path sensitization (a technique based on a powerful mathematical foundation - combinational logic), as well as recent developments in the following four areas of research were the motivation behind the research work of this thesis:

- Black-box test case design;
- Functional specification based testing;
- Software engineering tools and techniques; and
- Reliable design and fault-detection tests (fault detection by path sensitization)

The concepts from combinational logic based testing techniques such as fault detection by path sensitization have inspired our approach to test case design by "exercising" functional specifications represented in basic *canonical forms* (Section 3.2). Our investigation with regard to hardware path sensitization testing techniques (Section 3.1) shows that these techniques are more algorithmic and therefore easier to implement. However, their extension to software based testing (see Section 2.2), either suffers from being heuristic or not suitable for computerization (see Sections 5.2 and 5.3).

1.4 Summary of Main Contributions

In this thesis, we propose and evaluate an incremental sensitized test case design methodology for the design of non-redundant high-yield test requirements in a disciplined and systematic manner. We also implement an interactive prototype system to support and illustrate our methodology (DINSERT).

We demonstrate the feasibility of our approach by means of a prototype system implementation, called DINSERT. This prototype provides a user-friendly and attractive interactive test design environment. The system automatically constructs canonical forms of cause-effect graphs, generates test requirements based on sensitization of canonical forms, and incrementally generates test requirements. Redundant test requirements are automatically removed. We show how useful our approach is by applying the tool to TRP, comparing it with other effective approaches, and suggesting how to use our methodology in a development environment.

Applying such an approach to early stages of software development (inception, system-definition, system-design) has great potential for detecting ambiguities, inconsistencies, and even omissions - well before any target software is produced [Appl 83, Boeh 83, PrUr 84]. This thesis provides evidence that this methodology is an effective and efficient technique for producing more reliable specifications and thus, more reliable software.

1.5 Outline of the Thesis

In Chapter II, we briefly survey path sensitization techniques, starting with hardware testing techniques and concluding with extensions to software testing techniques.

In Chapter III, we thoroughly describe our incremental sensitized test case design methodology. We introduce fundamental concepts and basic definitions. Brief proofs and examples are used to clarify various aspects of our approach. We also present algorithms for the construction of canonical forms, generation of test requirements based on sensitization of canonical forms, and incremental generation of test requirements by redundancy removal. All algorithms are illustrated using a subset of the well-known TRP.

In Chapter IV, we describe in detail our INcremental SEnsitized Test Requirement Design (abbreviated DINSERT) prototype system. Here we discuss the various subsystems of DINSERT as well as the file environment, design issues, user interface etc. A complete description of how to use DINSERT appears in Appendix A. The Chapter ends with an assessment of the tool from the user and system points of view with discussion about its limitations and recommendations for future enhancements.

In Chapter V, we evaluate our methodology based on a comparison with other useful approaches such as Myers' heuristic "Cause-Effect Graphing" approach [Myer 79], and Walsh's "Equivalent Normal Form" [Wals 83]. Again, we compare them using the TRP.

Chapter VI contains the summary, conclusions, and suggestions for future research.

Chapter II
A BRIEF SURVEY OF SENSITIZATION TESTING TECHNIQUES

In this chapter, we sketch a number of hardware testing techniques based on sensitizing the system under test. Our aim is to reveal particular leads of logic faults *stuck-at-0* and *stuck-at-1*. With this background, we describe a software testing technique developed from some of these techniques, called Cause-Effect Graphing (abbreviated CEG).

This chapter's objective is to present an outline of related hardware testing techniques used for the diagnosis of combinational networks. For more details, the interested reader may consult standard works [Arms 66,72, Roth 66,67, ScDi 68, Chan 70, KoDe 71, Chap 74, ChCh 74, Bett 77, Muro 79, McCL 86].

2.1 Hardware Path Sensitizing Testing Techniques

Hardware tests are intended to verify that a product has no faults. There are two main approaches, probabilistic and deterministic. Probabilistic generation of tests involves no a priori knowledge of the product. The effectiveness for detecting faults is measured with a fault-simulator by "injecting" faults into the product [ChCh 75, Arms 72]. Deterministic test generation selects a special fault, then generates the tests that

detect that fault. Any other faults detected by the same test are not subsequently tested for.

Most test generation algorithms use path sensitization. Originally, a single path was sensitized for the origin of an assumed fault to the primary output [Arms 66]. But single path sensitization algorithms are not always successful even when the fault is testable. So Roth proposed the D-algorithm that sensitizes single or multiple paths [Roth 67]. This is a sophisticated version of the path sensitization method. Roth proved that, unlike the path sensitization method, the D-algorithm can find a test pattern if one exists considering multiple sensitized paths. In this sense the D-algorithm is the first test generation method proved to be algorithmic. But the D-algorithm needs longer test time and more memory space than the path sensitization method. Together the path sensitization and the D-algorithm are often called the *D-algorithm* [Muro 79].

2.1.1 Logic Faults: Stuck-at-0 and Stuck-at-1

Logic faults are faults that produce some deviation from the expected logical behavior of the circuit. So component failures that affect voltage, current, shapes of pulses or delays in the circuit, but do not alter the logical function are not logic faults.

The kind of faults you consider when testing a logic circuit depends on the kind of circuit you are testing. However, most of the faults in currently used circuits like diode-resistor circuits (DR), diode-transistor logic circuits (DTL), transistor-

resistor logic circuits (TRL) and transistor-transistor logic circuits (TTL) are represented by a gate being *stuck-at-0* or *stuck-at-1*. That is, an input or output may assume a fixed value, independent of the input applied to the circuit.

The truth table method provides a tool for the determination of a minimal set of fault-detection tests for combinational logic circuits. But this method is not practical (Section 2.1.2.1). So here we introduce a different design philosophy, based on the path sensitization technique. The two sections after that (2.1.2.2 and 2.1.2.3) have more to say on this method. For more information refer to [Arms 66,72, Muro 79].

The main idea behind the path sensitization procedure will be illustrated here by devising a test for detecting the following logical faults: suppose that we want to detect whether one of the inputs of gate 2, x_2, is permanently fixed to the value y (either *1* or *0*) by a fault in the NAND gate network in Figure 2.1 on page 23. When y is fixed to *0*, it is said to be "*stuck-at-0*". When y is fixed to *1*, it is said to be "*stuck-at-1*".

Now fixing some of the inputs to a network so that the network output depends on the signal on a particular lead is called "*sensitizing* the output to a particular lead". Just one sensitization pattern is usually satisfactory. It is possible to find a sensitization pattern by tracing a *sensitized path* through the network. All of the automatic test pattern generation programs are based on tracing sensitized paths through a network [McCL 86]. The path sensitization method and most other methods

(described in the next sections) are effective only when a single fault that is *stuck-at-1* or *stuck-at-0* is to be detected [FrMe 71, Frie 86].

Suppose that all other inputs to the gates through which signal y goes to be one of the network outputs are set to *1* (if we have a network of NOR gates, they are set to *0*) by setting inputs x_1, x_3, and x_4 to their proper values, as shown in Figure 2.1. We can do that by setting $x_1 = 0$ and $x_3 = x_4 = 1$ (e.g., gates 2, 4, and 6 have inputs of *1*).

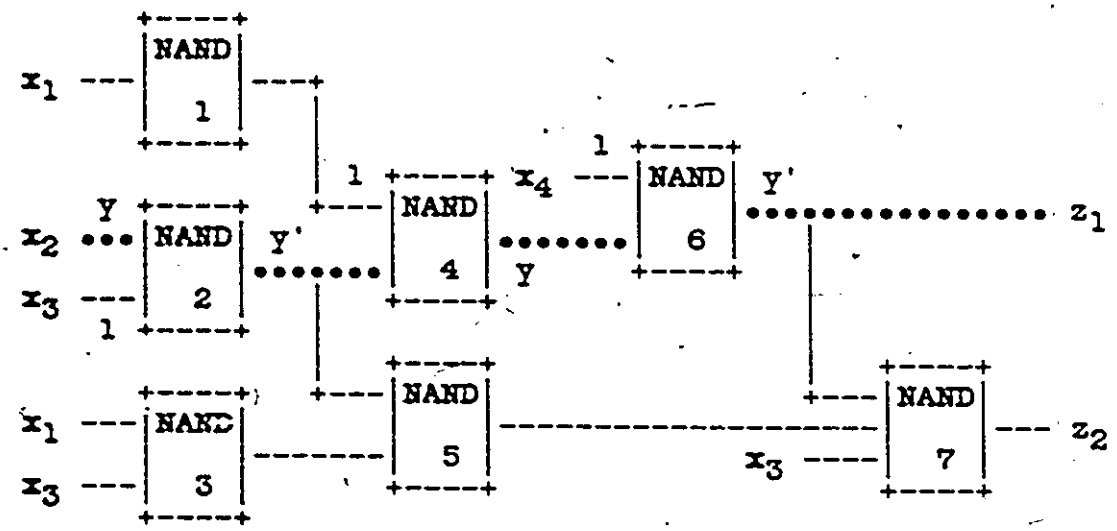


Figure 2.1: A Logic Network Describing a Sensitized Path and the Stuck-at-0 and Stuck-at-1 Sensitization.

The network output z_1 becomes y' (in a general network, y or y'). This path is called a *sensitized path* from input x_2 of gate 2 to network output z_1. So when the value of input x_2 to gate 2 is

set to a value different from the *stuck-at value*, y, we can find out whether y is *stuck-at-1* or *stuck-at-0*. In other words, if we set $x_2 = 1$, $x_1 = 0$ and $x_3 = x_4 = 1$, we can find out that y is *stuck-at-0* if z_1 shows output value *1*. This is because z_1 must be *0* for $x_1 = 0$ and $x_2 = x_3 = x_4 = 1$ if the network is not faulty. Similarly, when we set $x_2 = 0$, $x_1 = 0$ and $x_3 = x_4 = 1$, we can find out that y is *stuck-at-1* if $z_1 = 0$.

2.1.2 Derivation of Tests

In this section, we discuss several methods of deriving tests for a given fault in a combinational circuit. We assume that the circuit is non-redundant and so can have one fault at a time. First, we shall consider only faults where any wire in the circuit is *stuck-at-0* or *stuck-at-1*.

2.1.2.1 Truth Table Method

The most obvious method of deriving tests for a particular fault is by comparing the truth tables of the normal and the faulty circuits. Let the inputs to a combinational circuit be $(x_1, x_2, \ldots, x_n)$. And let its outputs be $(z_1, z_2, \ldots, z_m)$, where $z_i = f_i(x_1, x_2, \ldots, x_n)$, $i = 1,2,\ldots,m$. For any set of faults F, and any fault "a" *"a"-belongs-to-F* (i.e., by "a" we denote any faulty wire or path). Let $z_i^a = f_i^a(x_1, x_2, \ldots, x_n)$ be the value of the ith output when the fault is present. Then an input vector $x^j = (x_1^j, x_2^j, \ldots, x_n^j)$ is a test for detecting the fault "a" if and only if:

$f_i(x^j)$ XOR $f_i^a(x^j) = 1$, for some i: $1 <= i <= m$

Where XOR represents the "exclusive-OR" operator. The above equation implies that the fault "a" is detected by applying the input x^j and observing the output z_i. All the tests which detected any given fault can be obtained in a straightforward manner by comparing the truth tables of the normal and faulty circuits, as shown in an example in Figure 2.2.

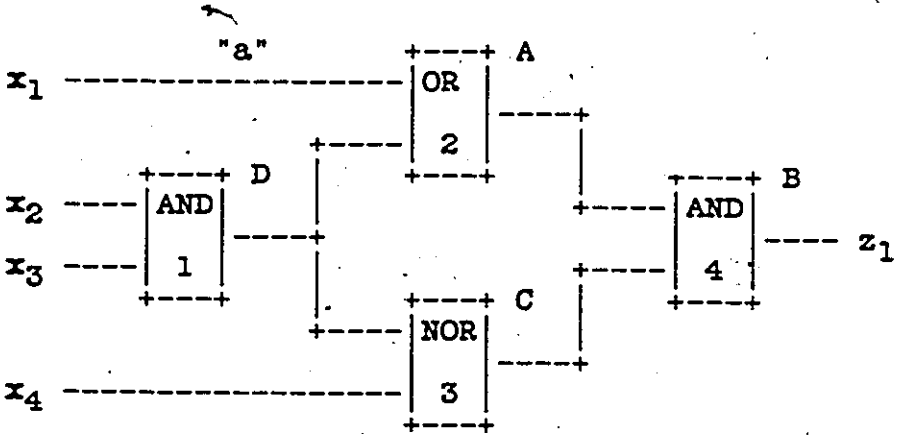


Figure 2.2: A Logic Circuit to Illustrate the Truth-Table Method. Let wire "a" be *stuck-at-1*. The truth tables of the normal and faulty circuits are shown in Table 2.1 on page 26.

Table 2.1 on page 26, shows that these three tests detect the "a" *stuck-at-1*:

$(x_1.x_2.x_3.x_4)$ = (0,0,0,0); (0,0,1,0,); (0,1,0,0)

Often it is convenient to represent tests by the corresponding minterms[1]. You can write the set of tests in the example taken from Figure 2.2 as $x_1'.x_2'.x_3'.x_4'$, $x_1'.x_2'.x_3.x_4'$ and

[1] A *minterm* is a product of all the variables of the function, where each variable may appear in the complemented or uncomplemented form [Koha 70].

$x_1'.x_2.x_3'.x_4'$. From the first two tests, we see that $x_1 - x_2 - x_4 - 0$ is a test independent of the value of x_3. Similarly, $x_1 - x_3 - x_4 - 0$ is a test independent of the value of x_2. The test can be represented by the Boolean expression: $x_1'.x_2'.x_4' + x_1'.x_3'.x_4'$. Test for detecting other faults in the circuit can be determined in a similar way.

The example shows that the method would be impractical even for circuits of moderate size, because a truth table will have to be constructed for every possible fault. We shall now discuss some methods of deriving tests without constructing these truth tables.

Table 2.1: Illustrating the Truth-Table Method.

x_1	x_2	x_3	x_4	z_1	z_1^a	z_1 XOR z_1^a
0	0	0	0	0	1	1
0	0	0	1	0	0	0
0	0	1	0	0	1	1
0	0	1	1	0	0	0
0	1	0	0	0	1	1
0	1	0	1	0	0	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	1	1	0
1	0	0	1	0	0	0
1	0	1	0	1	1	0
1	0	1	1	0	0	0
1	1	0	0	1	1	0
1	1	0	1	0	0	0
1	1	1	0	0	0	0
1	1	1	1	0	0	0

2.1.2.2 Path Sensitizing Applied to a Single Path

The conditions for propagating a change to an output, along any path can be determined from a knowledge of the logic circuit. This is done by assigning input values to each gate along the chosen path, so that its output depends on one particular input only. The conditions required for a change in one of the inputs to a gate to cause a change in its output depend on the type of gate involved. For AND and NAND gates, all inputs except the changing one should be 1. For OR and NOR gates, these inputs should be set at 0.

Returning to logic circuit of Figure 2.2 on page 25, $x_1 = 0$ in all the tests. For the output of the OR gate (number 2), to be sensitive to any change on wire "a", the other input to this gate should be 0. This means that, with the AND gate (number 1) $x_2 \cdot x_3 = 0$. For z_1 to be sensitive to any change in "a", the other input to AND gate (number 4), should be 1, and the output of the NOR gate (number 3) should be 1. This, in turn, requires that $x_4 = 0$. The three tests derived in example satisfy these conditions.

Path sensitizing general procedure: The general procedure for deriving tests using *path sensitizing* is that the faulty wire is assigned a value opposite to the fault condition. That is, a value of 1 is assigned to a wire with a *stuck-at-0* fault and vice versa for a *stuck-at-1* fault. A path is chosen from the fault to one of the output terminals. The inputs to the gates along this path are assigned values so as to propagate

any change on the faulty wire along the chosen path to the output terminal. The path is now said to be "*sensitized*". One or more tests for detecting the particular fault are obtained by determining network inputs that will produce the desired values on the path inputs along the sensitized path. To do this trace back from the gates along the sensitized path towards the network inputs and assign values to as many inputs as you need to get the desired signal values in the circuit.

This procedure may not yield a unique set of inputs for sensitizing a particular path. An arbitrary choice is made wherever different possibilities exist. If a contradiction is encountered, the process is repeated with a different choice, if such a choice exists. Otherwise, you may have to select a different path for sensitizing. If a consistent input combination is obtained, some inputs in it may be unspecified. This means that the test is independent of those inputs and may be assigned any values desired to minimize the test set. The example of the logic circuit in Figure 2.3 on page 29, shows how to do this.

In Figure 2.3, let the output of the NOR gate (number 2) be *stuck-at-0*. Referring to the outputs of gates by the names/numbers of the gates, we require B = 1 in order to detect the *stuck-at-0* fault. Let us arbitrarily choose to sensitize the path BDE to the output z_1. To sensitize this path we need A = 0 and C = 1. But B = 1 implies $x_2 = x_3 = 0$, which also makes A = 0. To make C = 1, we need $x_3 = x_4 = 1$ leading to a contradiction in the assignment of x_3. Since no arbitrary choices were made during

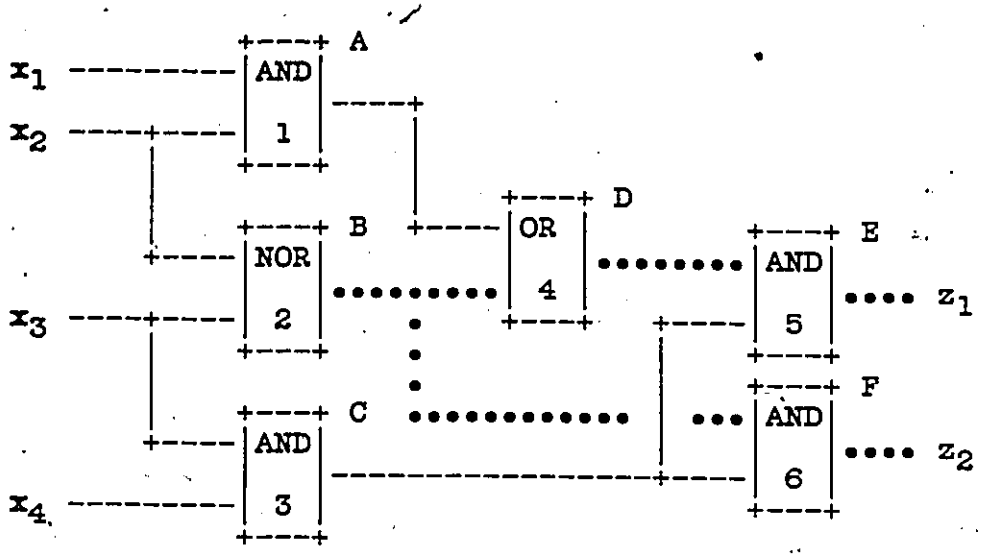


Figure 2.3: Single Path (one at a time) Sensitization.

the assignment of input values, and BDE is the only path from B to z_1, it follows that *stuck-at-0* fault at B cannot be detected at the output z_1. However, the fault can be detected at z_2 since $x_2 = x_3 = 0$ sensitizes the path BF and also makes B = 1. Inputs x_1 and x_4 are unspecified for this test.

2.1.2.3 Path Sensitizing Applied to Simultaneous Paths

The path sensitizing technique, as we have discussed it so far, attempts to sensitize only one path in the logic circuit at a time. The example in Figure 2.4 on page 30, shows why this procedure is inadequate. In the logic circuit of Figure 2.4 on page 30, let us try to derive a test for detecting the fault "a" *stuck-at-0* by sensitizing the path ABJ.

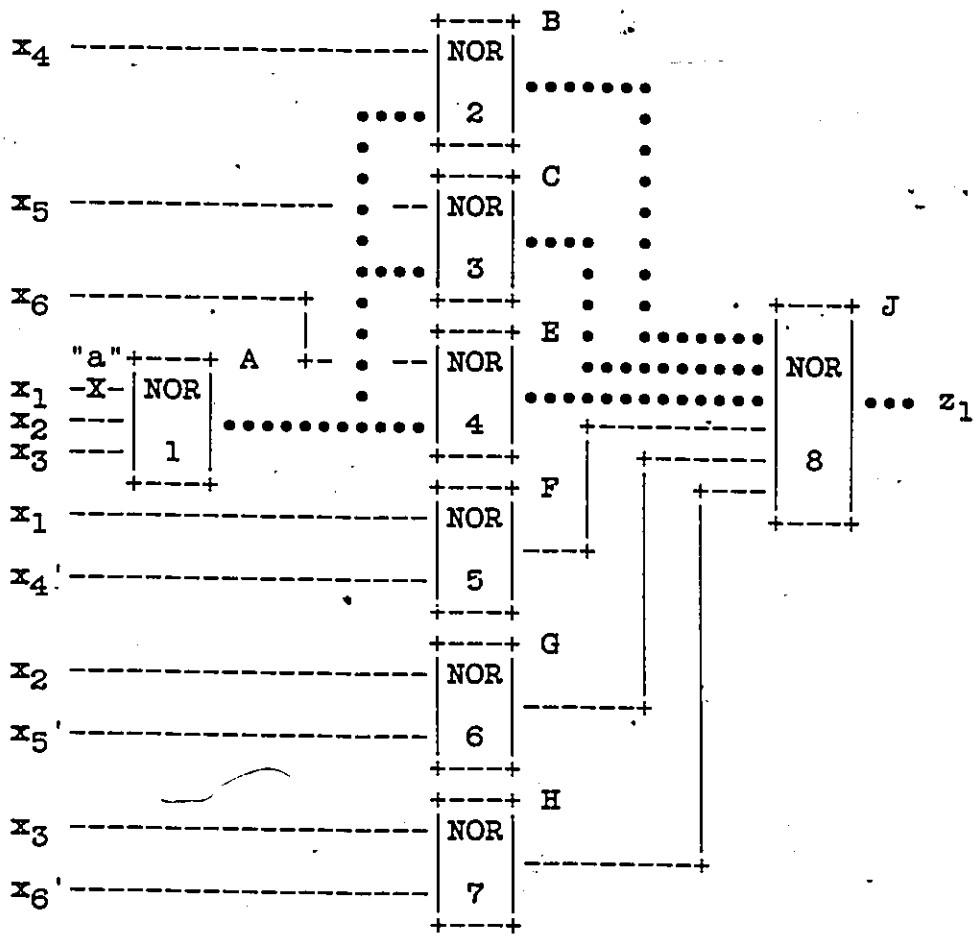


Figure 2.4: Simultaneous Path Sensitization.

First we set $x_1 = 1$ and $x_2 = x_3 = 0$ so as to apply a signal opposite to the faulty value to "a" and propagate the effect of the fault through gate A (number 1). Setting $x_4 = 0$ carries it through gate B (number 2). To propagate it through gate J (number 8), we require C = E = F = G = H = 0. Setting $x_5 = x_6 = 1$ to make C = E = 0, we find that G = H = 1 and we are unable to propagate the signal through J. Similarly, we can show that it is

impossible to sensitize the single paths ACJ or AEJ. But $x_4 = x_5 = x_6 = 0$ sensitizes the three paths simultaneously and also makes F = G = H = 0. So three inputs to gate J change from 0 to 1 as a result of fault "a" *stuck-at-0*, while the remaining three inputs are fixed at 0. The fault will cause the J's output to change from 1 to 0 and $x_1.x_2'.x_3'.x_4'.x_5'.x_6'$ is a test for "a" *stuck-at-0*. Similarly, if we set $x_1 = x_2 = x_3 = 0$ to detect "a" *stuck-at-1*, the output of gate A will change from 1 to 0 as a result of the fault. Here again, the three paths ABJ, ACJ, and AEJ have to be sensitized simultaneously by setting $x_4 = x_5 = x_6 = 0$ to detect the fault.

The above example shows the necessity of sensitizing more than one path, in deriving tests for certain faults. To sensitize more than one path, we need some method of identifying signals whose values depend on the fault. This is the main idea behind the D-algorithm [Roth 66].

2.1.2.4 The D-Algorithm

We first present an *informal version* of the *D-algorithm*, which may be considered to be path sensitization with the fault-dependent signals uniquely identified. Let us use the symbol D to represent a signal that is 1 in the normal (fault-free) circuit and 0 in the faulty circuit. As well, the symbol D' will be used to represent the signal that is normally 0, but becomes 1 when the fault is present.

Note that the definition of D and D' could be interchanged, but this interchange should be consistent through the circuit.

Thus, all Ds in the circuit imply the same value whether 0 or 1 and all D's will have the opposite value. With this meaning associated with D, the logical operations shown in Table 2.2, are easily verified.

Table 2.2: AND and OR definitions for the D-Algorithm.

+	0	1	D	D'
0	0	1	D	D'
1	1	1	1	1
D	D	1	D	1
D'	D'	1	1	D'

(left) OR

.	0	1	D	D'
0	0	0	0	0
1	0	1	D	D'
D	0	D	D	0
D'	0	D'	0	D'

(right) AND

EXAMPLE: The logic circuit shown in Figure 2.4 on page 30 is repeated in Figure 2.5 on page 33. To derive a test for "a" *stuck-at-0*, we assign a D to the line "a" and try to propagate the D to the output.

Setting $x_1 = 1$, applies a 1 to the wire "a". Then, $x_2 = x_3 = 0$ cause the output at gate A (number 1) to be D'. We arbitrarily choose to sensitize the path ABJ, by setting $x_4 = 0$ and also note that gates C and E both have a D' on one input. Attempting to make all inputs to gate J except B to 0 results in contradiction. However, if we set $x_5 = x_6 = 0$, we have B = C = E = D and F = G = H = 0. Now j = D' and "a" *stuck-at-0* is detected. The signal value in the circuit for this test are shown in Figure 2.5 on page 33.

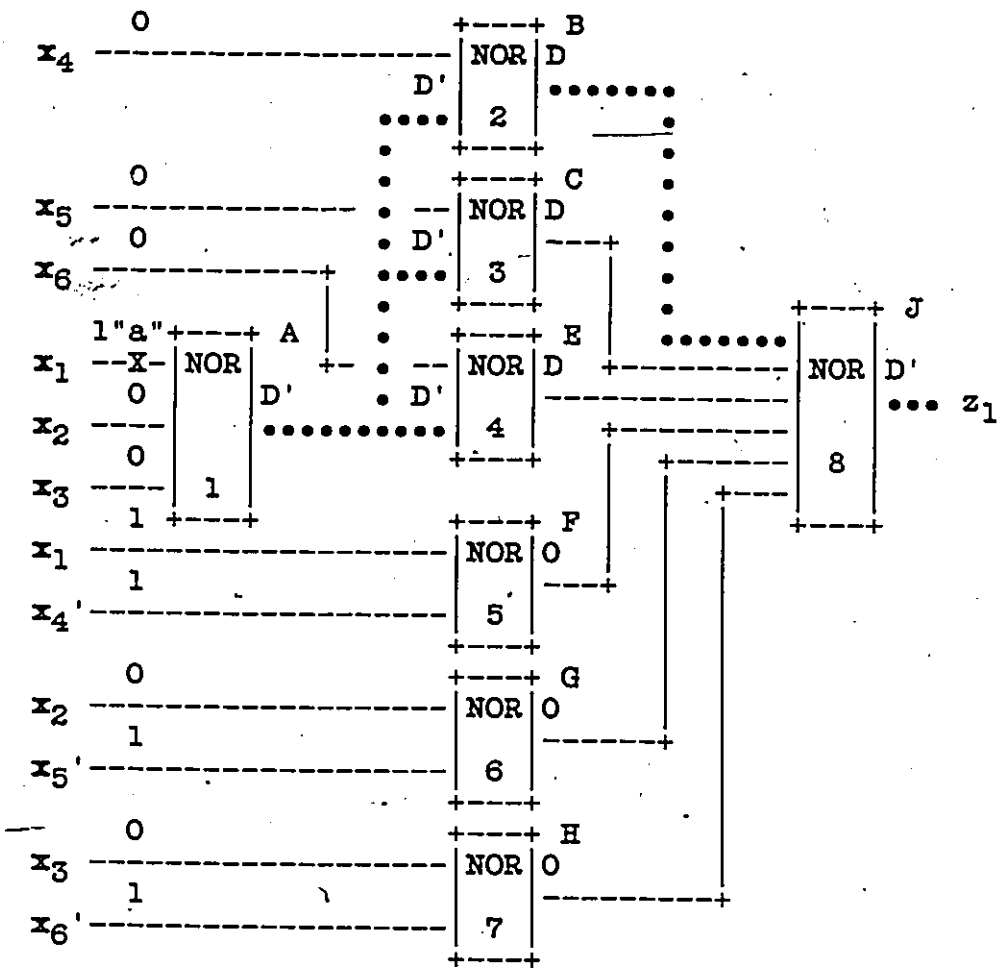


Figure 2.5: A Circuit to Illustrate an Informal Version of the D-Algorithm.

In the *formal D-algorithm*, a function of n variables $f(x_1, x_2, \ldots, x_n)$ is represented as a function of $(n+1)$ variables $g(x_1, x_2, \ldots, x_{n+1})$, such that $g(a_1, a_2, \ldots, a_{n+1}) = 1$ if and only if $a_{n+1} = f(a_1, a_2, \ldots, a_n)$. A set of prime implicants[2] covering

[2] A *prime implicant* of a function is an implicant such that any proper subset of its variables does not form an implicant of the function. In turn, if a function is expressed in the *sum-of-products* form, each *product-term* in the expression is called an *implicant* of the function.

the *function g* represents the truth table of the *function f* in a concise way. The set of cubes defined by these prime implicants is called the *singular cover* of the *function f*.

EXAMPLE: As an example, a two input NOR gate and its singular cover are shown in Figure 2.6, where "-" denotes "don't care" values. The singular cover of the NOR gate states that its output will be 1 if both inputs are 0, and 0 if one of the inputs is 1.

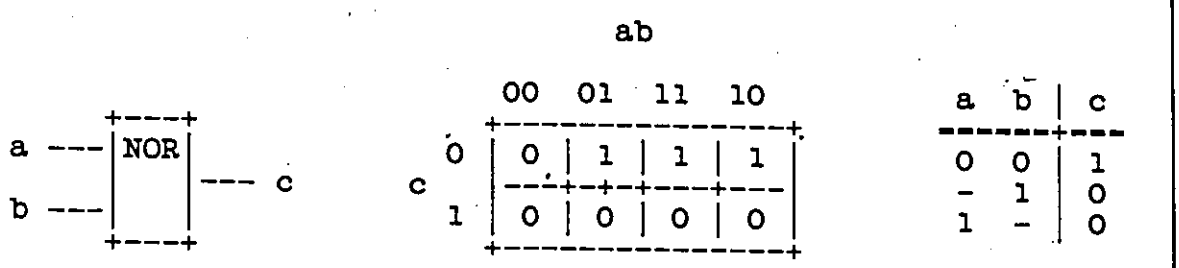


Figure 2.6: Singular Cover and Primitive D-cubes for a Two-Input NOR Gate.

In deriving tests for combinational circuits, it is useful to identify two special types of the inputs to a logic block. The first type are those inputs which cause the output of the block (assuming single-output blocks) to be different from its normal value, if a given fault is present in the block. These inputs are represented by the *primitive D-cubes* of the fault.

The second type of inputs, represented by the propagation *D-cubes* of a block, are those that cause the output of the block to depend only on one or more of its specified inputs and hence to propagate a fault at these inputs to the output [Chan 70].

For simple blocks, such as a NOR gate, and simple faults such as an input *stuck-at-1*, both types of D-cubes can be written down by inspection. For example, if the input lead a of the NOR gate of Figure 2.6 on page 34 is *stuck-at-1*, then $a = 0$, $b = 0$ will cause the output $c = 0$ if the fault is present and $c = 1$ otherwise. This is represented by the following *primitive D-Cube* of the fault:

a	b	c
0	0	D

where D represents the condition under which the correct output is 1 and the faulty output is 0. Note that this choice is arbitrary and the opposite choice could have been made.

On the other hand, if we are interested in propagating the effect of a fault (external to the particular gate) through the NOR gate of Figure 2.6 on page 34. Then the following D-cubes of the block are of interest, assuming that only one input to the gate may be faulty:

a	b	c
D	0	D'
0	D	D'

Here, the interpretation of the symbol D is slightly different. D may be 0 or 1, but all Ds in a D-cube always have the same value. D' always has a value complementary to D. The two propagation D-cubes of the NOR gate merely state that if one of its inputs is zero, the output is the complement of the other input. If multiple-input changes are to be propagated through the NOR gate, DDD' should be added to the list of propagation D-cubes.

Appendix C explains more about the D-algorithm (the original contribution of Roth [Roth 67]).

2.1.2.5 The Equivalent Normal Form

It often occurs that an assignment of values to the circuit inputs sensitizes several paths simultaneously. Moreover, in general, a path may be sensitized by assigning values only to a subset of the circuit variables. In many cases, this allows you to assign the remaining variables in such a way that several paths will be sensitized simultaneously in one test.

Unfortunately, there are no efficient direct methods for the selection of these tests. To use these properties and to find a "good" (i.e., nearly minimal) set of tests, you need to first transform the circuit into its "Equivalent Normal Form" and then to derive these tests from the transformed circuit [Koha 70].

Armstrong has proposed a method for deriving a *nearly minimal* set of fault-detection tests, using the *Equivalent Normal Form* (abbreviated ENF), of the circuit [Arms 66]. He has conjectured, but not proved, that this method will yield tests for detecting all single faults in an *non-redundant circuit*. The ENF of a circuit represents the dependence of the output on the inputs - and the states of internal wires. However, some information about the type of fault that may be present on each wire is omitted.

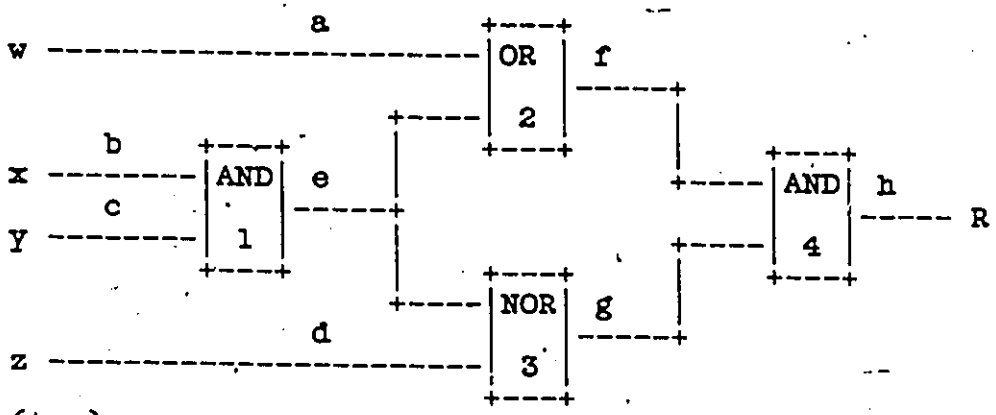
Procedure: The ENF of a circuit is obtained by expressing the output of each gate as a function of its inputs and preserving the identity of each gate by a suitable subscript

[Arms 66, FrMe 71]. Thus, if the inputs to an AND gate are labeled *a* and *b*, and the output is labeled *c*, we represent this by $c - (a.b)_c$

Applying this procedure to all gates in the circuit, we obtain an expression (usually in a factored form) of the output in terms of the input variables. This factored expression is then expanded into the sum-of-products form in the usual manner except that a pair of parentheses is removed, the subscript associated with it is concatenated to the subscripts of the variables within the pair of parentheses. Each subscripted input variable in the ENF is referred to as a *literal*. If an input variable appears in the ENF with different subscripts, they are considered to be distinct literals. Thus, a literal in the ENF is similar to a literal proposition in Poage's output proposition and represents a path from the input to the output [Poag 63].

EXAMPLE: We shall illustrate the procedure for deriving the ENF for the circuit of Figure 2.2 on page 25, which is repeated in the (top part) of Figure 2.7 on page 38.

An appearance of a literal in the ENF is usually tested for *stuck-at-1* by assigning the value of 0 to it, 1's to all other literals in the term and suitable values to additional variables so as to cause all other terms in the ENF to be 0. An appearance of a literal is tested for *stuck-at-0* by assigning the value of 1 to all literals in the term containing it and making all other terms equal to 0. Alternatively, an appearance of the same literal in the complemented ENF may be tested for *stuck-at-1*. In



(top)

$ENF(R) = (f.g)_h$

$= ((a+e)_f.(d'.e')_g)_h$

$= (((w)_a+(b.c)_e)_f.((z')_d.(d'+c')_e)_g)_h$

$= (((w)_a+((x)_b.(Y)_c)_e)_f.$

$((z')_d.((x')_b+(Y')_c)_e)_g)_h$

$= ((w_a+(x_b.Y_c)_e)_f.(z'_d.(x'_b+Y'_c)_e)_g)_h$

$= ((w_a+x_{b-e}.Y_{c-e})_f.(z'_d(x'_{b-e}+Y'_{c-e}))_g)_h$

$= ((w_{a-f}+x_{b-e-f}.Y_{c-e-f}).$

$(z'_{d-g}.x'_{b-e-g}+z'_{d-g}.Y'_{c-e-g}))_h$

$= w_{a-f-h}.x'_{b-e-g-h}.z'_{d-g-h} +$

$w_{a-f-h}.Y'_{c-e-g-h}.z'_{d-g-h} +$

$x_{b-e-f-h}.x'_{b-e-g-h}.Y_{c-e-f-h}.z'_{d-g-h} +$

$x_{b-e-f-h}.Y_{c-e-f-h}.Y'_{c-e-g-h}.z'_{d-g-h}$

(bottom)

Figure 2.7: Illustration of the ENF. Top, the logic circuit. Bottom, procedure for deriving the corresponding ENF.

the example circuit (top) of Figure 2.7, the literal w_{a-f-h} in

the first term can be tested for *stuck-at-1* by the input combinations w'.x'.z' and for *stuck-at-0* by w.x'.y'.z'.

Armstrong has shown that any test for a literal appearance in the ENF also sensitizes the path in the original circuit represented by the subscripts of the literal. So a test for a literal will test the wires represented by its subscripts for *stuck-at-1* or *stuck-at-0*. A set of tests for detecting all *stuck-at-1* and *stuck-at-0* faults in the circuit can be obtained if it were possible to:

1. Select a set of literals such that their subscripts cover all wires in the circuit.
2. Derive a set of tests which test at least one appearance of each such literal in the ENF (or the *complemented-ENF* [Arms 66]) for *stuck-at-1* and *stuck-at-0*.

Alternatively, each literal in both the ENF and its complemented form may be tested for *stuck-at-1* (or *stuck-at-0*). The set of tests so obtained may contain more than one test for each fault, because the same wire may be in more than one path sensitized by the sets of tests. Also note that testing every literal in the ENF for both *stuck-at-1* and *stuck-at-0* (if this is possible) is sufficient for testing the entire circuit but is not necessary. This leads us to the following conjecture by Armstrong [Arms 66]:

> "*Testing every testable literal in the ENF of a circuit for both stuck-at-0 and stuck-at-1 is sufficient for detecting all stuck-at-0 and stuck-at-1 faults in an non-redundant circuit*".

If the above conjecture is true, certain terms in the ENF may be omitted, since they cannot be used for testing any literal. If an input variable and its complement appear in the same term of an ENF, but with different subscripts, that term may be omitted. Such terms in the ENF correspond to two (or more) fanout paths, where the number of inversions along one path is even and along another is odd. Unless a fault is present in one of the fanout branches, this term will always have the value of 0 and therefore will have no effect on the derivation of tests for literals in other terms. If terms containing complementary variables are discarded, the reduced ENF so obtained may not contain literals corresponding to some connections in the circuit.

It has *not* been shown that the faults on these connections will be detected by tests derived for other literals in the reduced ENF, but no counterexample has been found. The ENF for the circuit of Figure 2.7 on page 38, can be reduced to:

The "reduced ENF" for the circuit is:

$$ENF(R) = w_{a-f-h} \cdot x'_{b-e-g-h} \cdot z'_{d-g-h} + w_{a-f-h} \cdot y'_{c-e-g-h} \cdot z'_{d-g-h}$$

The "complemented ENF" for the circuit is:

$$ENF(R') = w_{a-f-h} + x_{b-e-g-h} \cdot y_{c-e-g-h} + z_{d-g-h}$$

Though the ENF method is simpler that Poage's method [FrMe 71] of deriving tests, the latter has the advantage that tests can be derived for multiple faults. Both methods are practical only for

relatively small circuits. For larger circuits, the path sensitizing technique or the D-algorithm or a combination of these methods, seems to be preferable.

2.1.3 Minimization of Test Sets

The methods discussed in the preceding sections are useful for deriving a set of tests for detecting any fault in an non-redundant combinational circuit. The set of tests so obtained may be much larger than necessary for detecting all faults. So a practical problem of these methods is obtaining a minimal or near-minimal set of fault-detection tests. Another problem of interest is that of locating the fault in a circuit to a gate. This is the *diagnosis problem.*

Tests may be applied to a circuit using a *fixed (combinatorial) schedule* or an *adaptive (sequential) schedule* of testing. In fixed schedules, the order in which tests are applied is irrelevant. Fixed schedules are particularly suited for fault detection in circuits where all faults are equally likely to occur. In adaptive schedules, the next test to be applied is determined by the outcome of earlier tests. Adaptive schedules may require fewer tests than fixed schedules for locating faults. They may also be useful for detecting faults when the faults are not equally probable. But here we are only interested in the fixed (combinatorial) schedules.

2.1.3.1 Near-Minimal Test Sets Using Equivalent Normal Form

Armstrong has proposed a heuristic method of obtaining a "near-minimal" set of tests without deriving the complete fault table [Arms 66]. Using the ENF of a circuit and a "scoring function", tests are derived, one at a time, so that those tests which are likely to be necessary in minimal test set are derived first. The scoring function is devised so as to take advantage of the following two properties:

1. The term containing the literal to be tested should contain the fewest number of variables.
2. The literal to be tested should be such that its complement appears the largest number of times in the remaining terms of the ENF.

The first property assures that several variables will remain unassigned while testing the chosen variable, thereby increasing the chances of being able to assign these variables so as to test other literals simultaneously and hence, increase the number of faults detected by the test.

When the chosen literal is tested for *stuck-at-1*, the second property causes 1's to be assigned to literals in other terms; also makes it more likely to be able to test simultaneously some other literal in one or more of those terms for *stuck-at-1*.

Recall that a literal in the ENF is tested for *stuck-at-1* by additional variables so as to cause all other terms in the ENF to be 0. Tests for literals *stuck-at-0* are derived by using the

complemented ENF. Instead of testing for *stuck-at-1* faults on literals in both the ENF and its complement, we may test them for *stuck-at-0*. Since all literals in a term will be tested for *stuck-at-0* simultaneously, it is sufficient to specify a *scoring function* for each term in the ENF and complemented ENF.

EXAMPLE: As an example of the derivation of a near-minimal set of fault-detection tests using Armstrong's method, consider the circuit[3] of Figure 2.8 on page 44; it involves four-variables *A*, *B*, *C*, and *D*, and one output, *T*. The gates are numbered *1 though 7*, and the internal connections are labeled by the letters *i*, *j*, *k*, *m*, *u*, and *v*.

Recall that the ENF of logic circuit is obtained by expressing the circuit output as a sum-of-products such that each path has a corresponding distinct product-term [Gill 76, Koha 70]. Thus, by tracing all paths from the circuit output of Figure 2.8 on page 44 to every circuit input while recording the gates through which the paths pass.

Starting with gate (number 7), at (Level-1), the subscript 7 identifies the gate through which inputs u and v pass. Thus, we obtain:

$$ENF(T) = (u+v)_7 \qquad \text{(Level-1)}$$

Proceeding to the second level of gates, (Level-2), we find

$$ENF(T) = ((B.j)_4+(k.m)_6)_7 \qquad \text{(Level-2)}$$

---

[3] This example is taken from Armstrong [Arms 66].

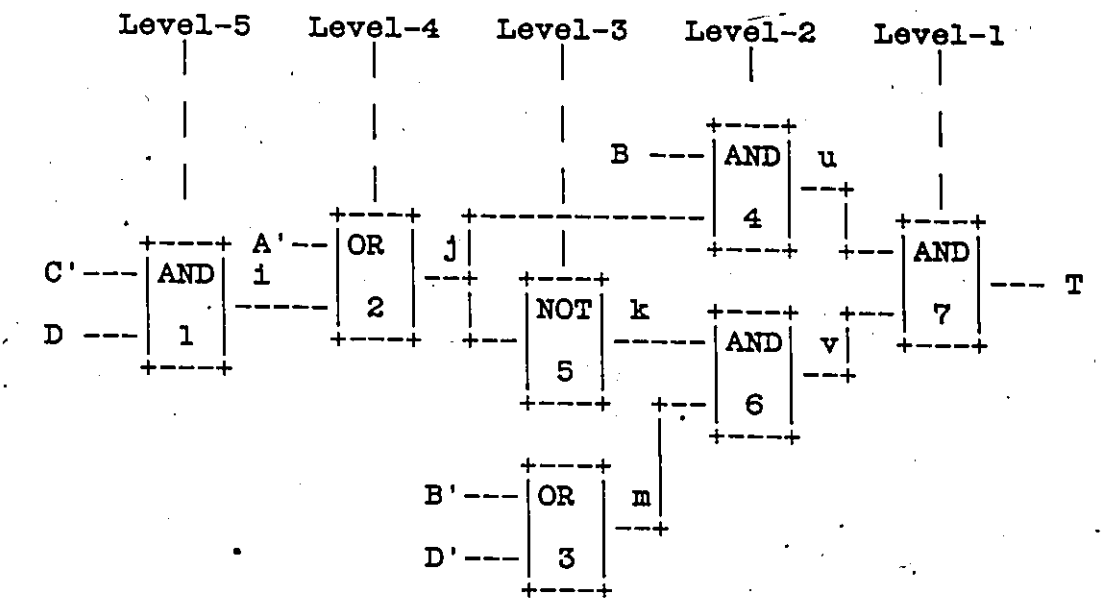


Figure 2.8: A Network to be Tested Transformed in ENF.

In a similar manner we expand T in terms of inputs to remaining gate levels:

$$ENF(T) = ((B.j)_4+((j')_5.(B'+D')_3)_6)_7 \quad \text{(Level-3)}$$

$$ENF(T) = ((B.(A'+i)_2)_4+(((A'+i)'_2)_5.(B'+D')_3)_6)_7 \quad \text{(Level-4)}$$

$$ENF(T) = ((B.(A'+(C'.D)_1)_2)_4+ (((A.(C+D')_1)_2)_5.(B'+D')_3)_6)_7 \quad \text{(Level-5)}$$

(Equation 1)

This expresses output T as a function of only the *external inputs*. The subscripts associated with each pair of parentheses is now "distributed" to all the terms within the parentheses, and as a result the *expression* for the ENF(T) is obtained as follows:

$$ENF(T) = A'_{2-4-7}.B_{4-7} + \quad \text{(Term-1)}$$

$$B_{4-7}.C'_{1-2-4-7}.D_{1-2-4-7} + \quad \text{(Term-2)}$$

$$A_{2-5-6-7}.C_{1-2-5-6-7}.B'_{3-6-7} + \quad \text{(Term-3)}$$

$$A_{2\text{-}5\text{-}6\text{-}7} \cdot C_{1\text{-}2\text{-}5\text{-}6\text{-}7} \cdot D'_{3\text{-}6\text{-}7} \; + \qquad \text{(Term-4)}$$

$$A_{2\text{-}5\text{-}6\text{-}7} \cdot D'_{1\text{-}2\text{-}5\text{-}6\text{-}7} \cdot B'_{3\text{-}6\text{-}7} \; + \qquad \text{(Term-5)}$$

$$A_{2\text{-}5\text{-}6\text{-}7} \cdot D'_{1\text{-}2\text{-}5\text{-}6\text{-}7} \cdot D'_{3\text{-}6\text{-}7} \qquad \text{(Term-6)}$$

(Equation 2)

Each variable of the ENF(T), together with its *subscripts*, specifies uniquely a path from the corresponding circuit input to the output. For example, $B'_{3\text{-}6\text{-}7}$ in Term-3 of Equation 2 specifies the path from input B' into gate (number 3), through gates (number 6, and 7) to the output (Figure 2.8 on page 44), while literal $A_{2\text{-}5\text{-}6\text{-}7}$ identifies the path from input A' into gate (number 2), through gates (number 2, 6, and 7) to the output. In general, if the number of inversion elements within a path (i.e., NOT, NAND, and NOR gates) is *odd*, the corresponding variables will appear in the ENF in a polarity "opposite" to the one they have in the circuit.

If the number of inversion elements is *even*, the polarity of the variable is "unaltered". Although the circuit of Figure 2.8 on page 44 is in minimal form, its corresponding ENF contains several "logically redundant" terms (e.g., Term-5 of ENF(T) Equation 2). Note also that, for the purpose of fault detection, the literals $D'_{1\text{-}2\text{-}5\text{-}6\text{-}7}$ of (Term-5 and Term-6) and $D'_{3\text{-}6\text{-}7}$ of (Term-4 and Term-6) are not equivalent, since they identify two different paths 1-2-5-6-7 and 3-6-7, respectively.

The *complement-ENF* of a circuit is obtained by complementing the ENF of that circuit [Arms 66]. Thus, the *complement-ENF(T')* for the running example is found by complementing (Equation 1), and expanding it, resulting in:

$$
\begin{aligned}
ENF(T') = {} & A'_{2-5-6-7} \cdot B'_{4-7} + && \text{(Term-1)} \\
& A_{2-4-7} \cdot A'_{2-5-6-7} \cdot C_{1-2-4-7} + && \text{(Term-2)} \\
& A_{2-4-7} \cdot A'_{2-5-6-7} \cdot D'_{1-2-4-7} + && \text{(Term-3)} \\
& B'_{4-7} \cdot C'_{1-2-5-6-7} \cdot D_{1-2-5-6-7} + && \text{(Term-4)} \\
& A_{2-4-7} \cdot C_{1-2-4-7} \cdot C'_{1-2-5-6-7} \cdot D_{1-2-5-6-7} + && \text{(Term-5)} \\
& A_{2-4-7} \cdot C'_{1-2-5-6-7} \cdot D'_{1-2-4-7} \cdot D_{1-2-5-6-7} + && \text{(Term-6)} \\
& B'_{4-7} \cdot B_{3-6-7} \cdot D_{3-6-7} + && \text{(Term-7)} \\
& A_{2-4-7} \cdot B_{3-6-7} \cdot C_{1-2-4-7} \cdot D_{3-6-7} + && \text{(Term-8)} \\
& A_{2-4-7} \cdot B_{3-6-7} \cdot D'_{1-2-4-7} \cdot D_{3-6-7} && \text{(Term-9)}
\end{aligned}
$$

(Equation 3)

So a *stuck-at-1* test for $A'_{2-5-6-7}$ in (Equation 3) is a *stuck-at-0* test for $A_{2-5-6-7}$ in (Equation 2).

Now consider the problem if testing by *path sensitization* the two level hypothetical circuit shown in Figure 2.9 on page 47. This corresponds to the ENF(T) (Equation 2) of Figure 2.8 on page 44. Since such a circuit consists of a number of AND gates, all in Level-two, feeding a single OR gate, at Level-one, in order to test a particular input for *stuck-at-1* fault, it is necessary to assign a value 0 to that input and a value 1 to the remaining gate inputs. In addition, it is necessary to "disable" other AND gates by ensuring that at least one input of each has the value 0. This procedure guarantees that unless the circuit is faulty, the output *T* of the OR gate will be 0.

Evidently, it may often happen that a single assignment of values is a test for two or more gate inputs. In fact, Armstrong describes a procedure which results in simultaneous tests for as

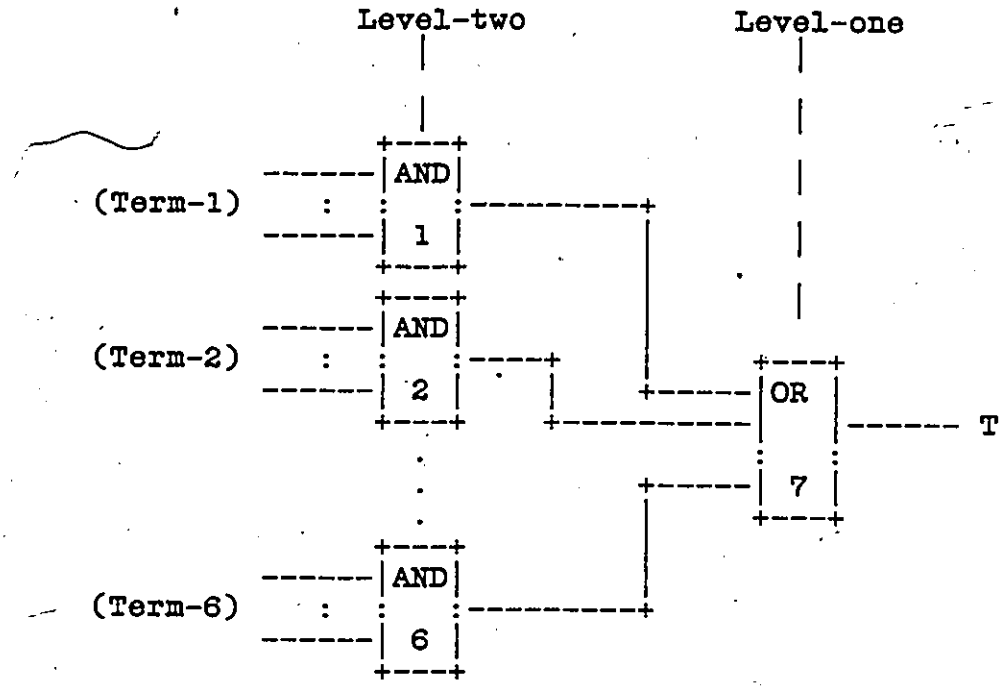


Figure 2.9: An Equivalent AND-OR Circuit Realizing a Multilevel Circuit. This AND-OR circuit may realize the ENF(T) of the multilevel circuit of Figure 2.8 on page 44.

many gate inputs as possible, similar to that of Roth's D-algorithm [Arms 66, Roth 67].

Clearly, if a 1 is assigned to all literals of a particular gate and at least one 0 is assigned to literals in each remaining gate, then all literals of that gate are tested for *stuck-at-0* faults.

Fault-detection tests for ENF's are easily constructed due to the simplicity and uniformity of their structures. All that is needed is to "select" a set of literals whose paths contain every connection in the corresponding hypothetical circuit (equivalent

AND-OR circuit), and to find a set of tests that check at least one appearance of each of these literals for *stuck-at-0* and *stuck-at-1* faults. This set of tests clearly detects any *stuck-at-0* or *stuck-at-1* faults in the ENF. Note that it is sufficient to test just a single appearance of each literal. For example, if B'_{3-6-7} of Equation 2 has been tested in Term-3, it is not necessary to test it again in Term-5.

2.1.4 Diagnosing Tree

The alternative representation of a set of tests in the form of the diagnosing tree is useful for obtaining near-minimal sets of diagnostic tests. The diagnosing tree is a directed graph whose nodes are tests and the outgoing branches from a node represent the different outcomes of the particular test. For a single-output circuit, there will be only two branches stemming from each node, corresponding to the success and failure of the test. Each branch is labeled with the possible sets of single faults that could be in the circuit, as determined by the tests already applied.

If combinational testing is to be used, the test to be applied at any stage of the testing is independent of the results of the past tests. In this case, the diagnosing tree can be arranged in levels and all nodes at the same level represent the same test. The diagnosing tree is continued until there is at most one fault associated with each branch. The problem of finding a minimal set of diagnostic tests becomes that of finding a diagnosing tree with the fewest number of levels.

Even though the number of tests required for diagnosis with combinational testing is independent of the order of tests, the number of levels in the diagnosing tree is dependent on the order of application of tests. This is caused by the masking of certain unnecessary tests by the particular order of testing as shown in the following example:

> ***EXAMPLE***: Let three tests t_1, t_2, and t_3 be used for distinguishing between four faults in a circuit. The diagnosing trees starting with t_1 and t_2 are shown in Figure 2.10 on page 50. The tests t_1, t_2, and t_3 dichotomize the faults as $(f_1; f_2f_3f_4)$, $(f_1f_2; f_3f_4)$, and $(f_1f_3; f_2f_4)$ respectively. In this example, the fact that t_1 is unnecessary is masked when it is applied first. In Figure 2.10 on page 50, "-O-" represents the "empty set".

To obtain a minimal set of diagnostic tests, it would be necessary to construct the diagnosing tree for every possible ordering of tests. Since this would be impractical except for very small problems, a heuristic method of choosing the ordering, leading to a good, but not necessarily optimal solution can be employed [Chan 65].

At each step, the test that will distinguish between the largest number of faults not already distinguished is applied. This is exactly what Armstrong's *scoring function* also tries to do - for fault detection only [Arms 66]. But in this function you construct a test with the desired property at each step, instead of choosing this test from a given set.

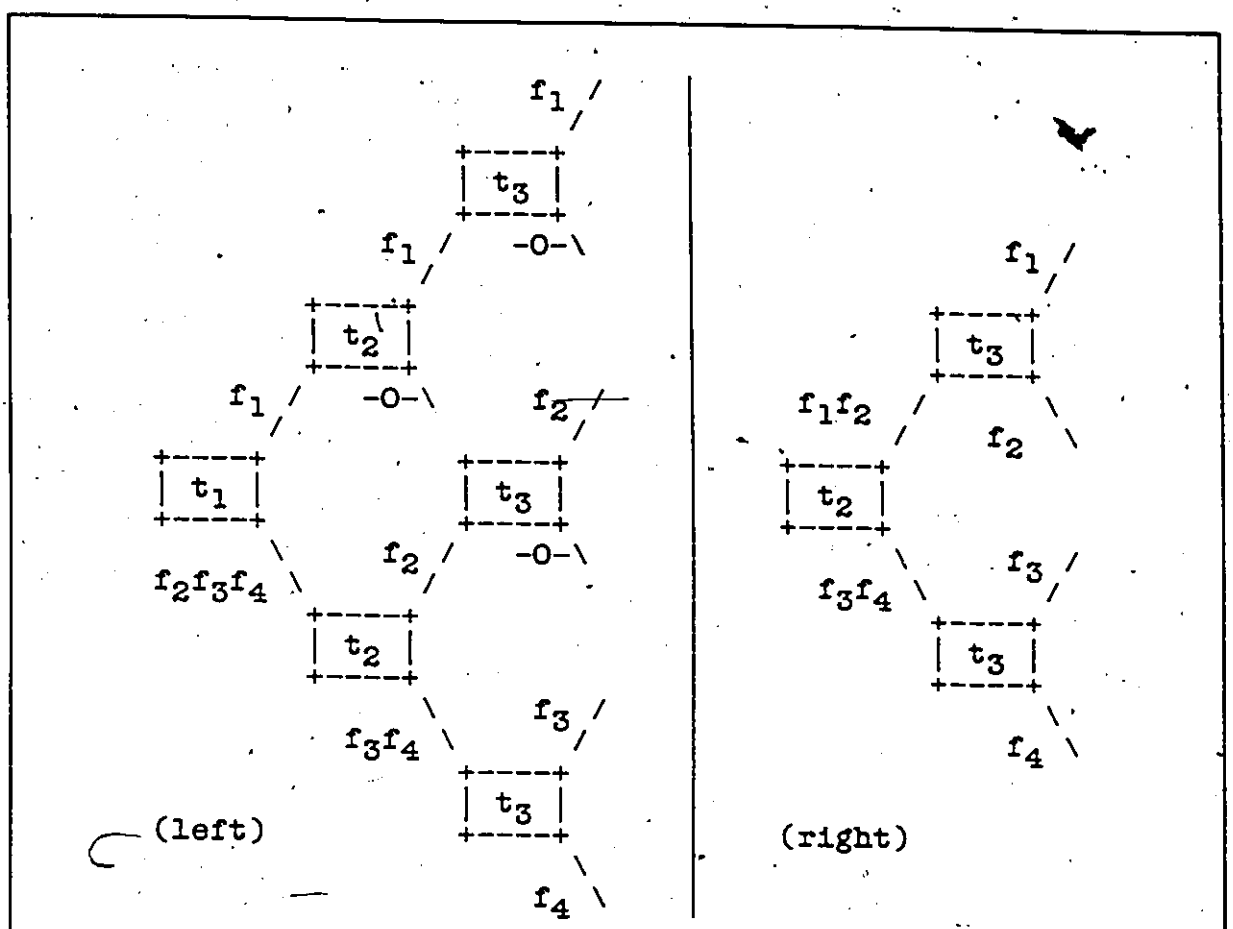


Figure 2.10: An Illustrative Diagnosing Tree.

2.2 Extension to Software Sensitization Based Testing Techniques

As we have seen in Section 1.1.3.2, Elmendorf proposed a technique for transforming a natural language specification into a *Cause-Effect Graphing* (abbreviated CEG) [Elme 73]. This technique comes mainly from *hardware logic testing*, and has become a well-known software test design method.

2.2.1 Cause-Effect Graphing

CEG is a black-box design technique that aids in selecting, in a systematic way, a high-yield set of test requirements. It explores *combinations* of specifications, and has a beneficial side effect in pointing out incompleteness and ambiguities in the specifications [Myer 79].

First, the semantic content of a functional specification is translated into a formal language (network representation). Through this network we can represent specifications as logical relationships between inputs (*causes*) and outputs (*effects*). This network is called a *Cause-Effect Graph*. Next, environmental and syntactic constraints are incorporated as annotations to this graph. The augmented graph is then analyzed to yield a set of test requirements. These requirements can then be organized as a limited-entry decision table. Each column in the table represents a test requirement and is used to guide the selection of actual test data.

This presentation addresses the transformation of functional specifications into CEG and the transformation of these graphs into a limited-entry decision table. The final step of selecting actual test data to satisfy test requirements is considered to be the most difficult step of this technique. However, since our focus in this thesis is test design, we do not treat this important issue of test implementation.

2.2.1.1 Specifying the System Under Test

The following process is used to represent the system under test as a *Cause-Effect Graph*:

1. The specification is divided into "workable" pieces. This is necessary because CEG becomes unwieldy when used on large specifications. For instance, when testing a time-sharing system, a "workable piece" might be the specification for an individual command. When testing a compiler, one might treat each programming-language statement individually.
2. The causes and the effects in the specification are identified:
   - A *cause* is a distinct *input condition* or an *equivalence class* of *input conditions*.
   - An *effect* is an *output condition* or a *system transformation* (a lingering effect that an input has on the state of the program or system).

   For instance, if a transaction to a program updates a master file, the alteration to the master file is a system transformation. A confirmation message would be an *output condition*. Each cause and effect is *assigned* unique number.
3. The *semantic content* of the specification is analyzed and *transformed* into a *Boolean graph linking* the *causes* and *effects*. This is the *Cause-Effect Graph*.

4. The *graph* is annotated with constraints on combinations of causes and/or effects that reflect reality because of syntactic or environmental constraints. A common example is the case of "exclusive" causes, e.g., the first position (i.e., column-1) may contain either a "1" or a "2". These causes are mutually exclusive.

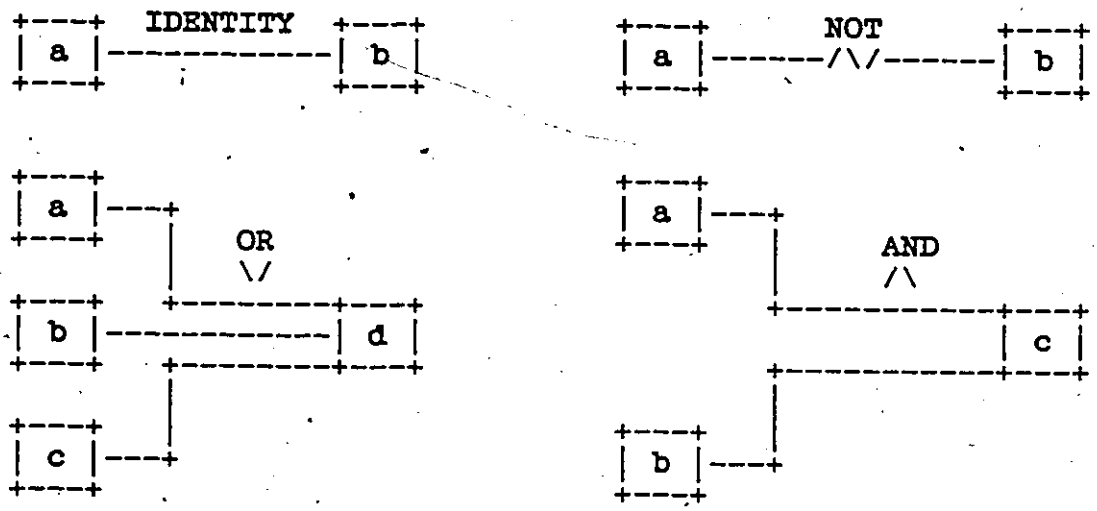


Figure 2.11: Basic Cause-Effect Graph Symbols.

Figure 2.11 illustrates the basic logical constructs for the Boolean graph. *Nodes* represent *causes*, *effects*, or *gates* (AND,OR,NOT). Each node has the value (0 or 1); 0 represents the "absent" state and 1 represents the "present" state. The Boolean connectives are shown in Figure 2.11.

- The IDENTITY function

```
If  (a .EQ. 1)  then
      b = 1
else
      b = 0
endif
```

- The NOT function

```
If  (a .EQ. 1)  then
      b = 0
else
      b = 1
endif
```

- The AND function

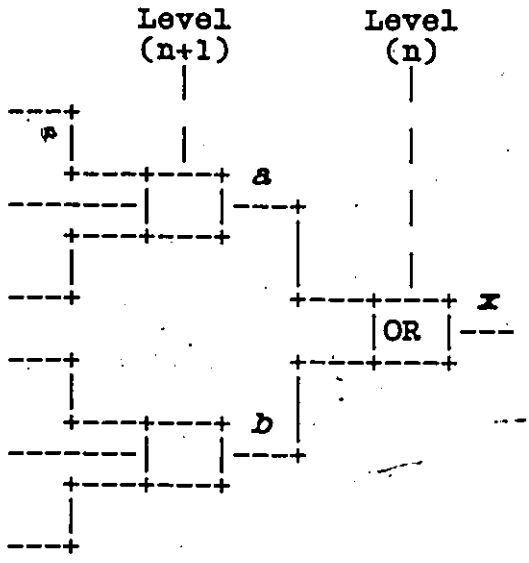
```
If  (a .EQ. 1  .AND.  b .EQ. 1)  then
      c = 1
else
      c = 0
endif
```

- The OR function

```
If  (a .EQ. 1  .OR.  b .EQ. 1  .OR.  c .EQ. 1)  then
      d = 1
else
      d = 0
endif
```

The last two functions, (AND,OR), can have any number of inputs (in practice, however, a fanin/fanout limit is set). Next, in Figure 2.12 on page 55, we summarize the CEG heuristic considerations, used by Myers for generating test requirements [Myer 79].

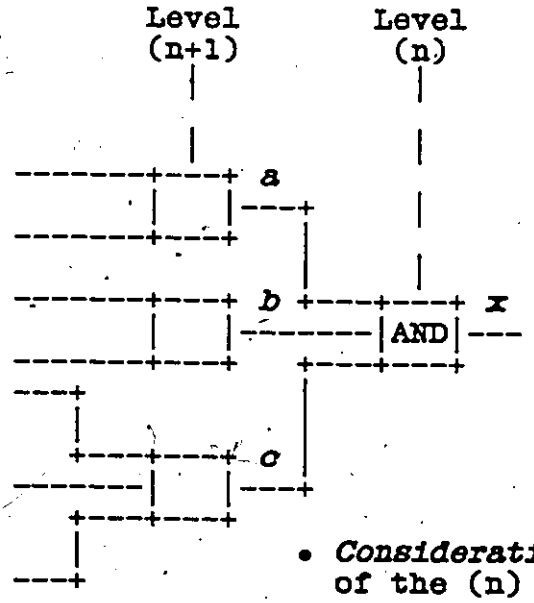
Sample Graph-Subsections



States / Situations
(with respect to n-Level)

a *b*	*x*	‹4-Situations›
0 0	0	‹1›
0 1	1	‹2›
1 0	1	‹3›
1 1	1	‹4›

- *Consideration-one:* If *x* is to be 1 (e.g., possible situations: ‹2›,‹3›,‹4›) do not include ‹4›.
- If *x* is to be 0, then enumerate all (n+1) Level situations (e.g., of *a* and *b* which thus leads to situation: ‹1›) in the (n) level.



a *b* *c*	*x*	‹8-Situations›
0 0 0	0	‹1›
0 0 1	0	‹2›
0 1 0	0	‹3›
0 1 1	0	‹4›
1 0 0	0	‹5›
1 0 1	0	‹6›
1 1 0	0	‹7›
1 1 1	1	‹8›

- If *x* is to be 1, then enumerate all situations of the (n+1) Level, leading to situation ‹8› in the (n) Level.
- *Consideration-two:* From range (‹2› to ‹7›) of the (n) Level, include one only.
- *Consideration-three:* If *x* is to be 0 (e.g., situation ‹1›, in the (n) Level) include only one situation in the (n+1) Level, so that ‹1› is satisfied.

Figure 2.12: Myers' CEG Trace Considerations.

2.2.1.2 Incorporating Environmental Constraints

Environmental and semantic considerations limit the feasible domain of inputs. *Physically impossible* or *logically contradictory* combinations of causes fall into this category. Such restrictions, called environmental constraints, can take any of the following forms:

- A set of causes may be *mutually exclusive*, meaning that at most one of the set can be invoked.
- A set of causes may be *all-inclusive*, meaning that at least one of the set must be invoked.
- A set of causes may be *both mutually-exclusive* and *all-inclusive*, meaning that one and only one of the set must be invoked.
- The *state* of one cause may require a certain state for another cause in an input pattern.

In addition, there is frequently a need for constraints imposed among effects. That is, the state of one effect may mask a certain state for another effect in the output pattern. Environmental constraints are an important aspect of CEG because they define practical limits on satisfying test case requirements. The graphical representations of these constraints are shown in Figure 2.13 on page 57.

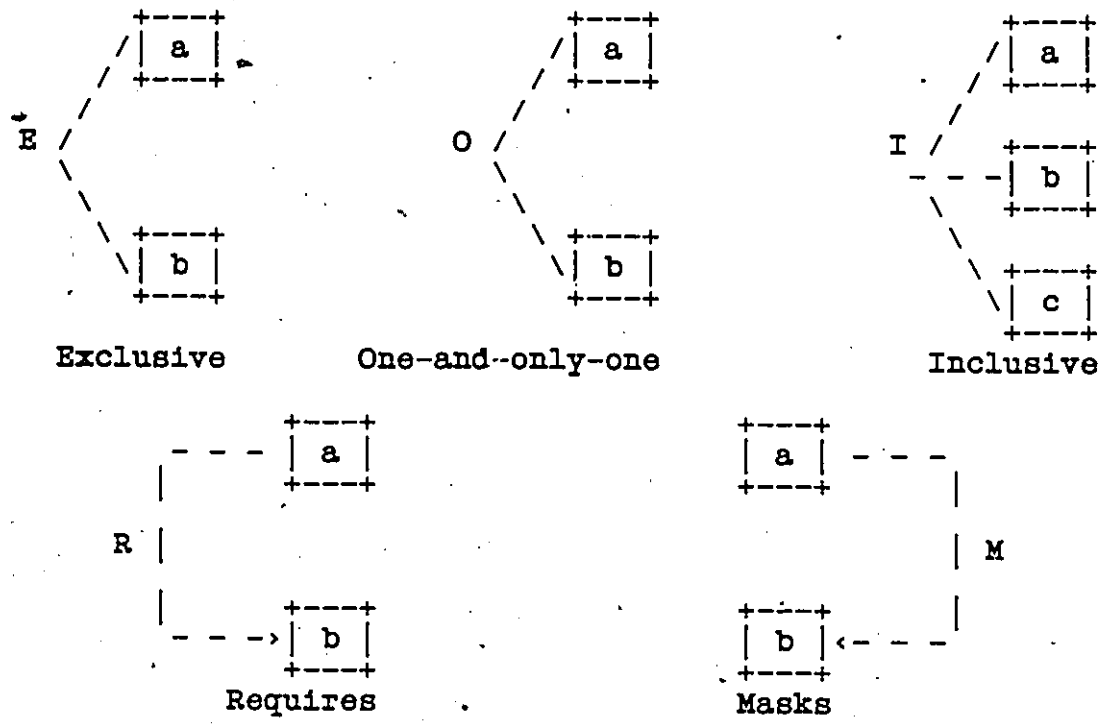


Figure 2.13: Cause-Effect Graph Constraint Symbols.

2.2.1.3 Transforming the Cause-Effect Graph into a Test Requirements Decision Table

The next step is the generation of a limited-entry decision table to represent test requirements. This is the most critical step in this technique. Myers [Myer 79] proposed a heuristic procedure comprising the following four steps:

1. Step-one: Select an effect to be 1. This denotes the "present" or "observable-state".
2. Step-two: Trace back through the graph, finding all high-yield combination of causes (constraints are used later to find real test data) that will set this effect to 1. To carry out this step, *three considerations* are employed:

a. Consideration-one: When you trace back through an OR node whose output should be 1, never set more than one input to the OR to 1 simultaneously. This is called *path sensitizing*. Its objective is to avoid not detecting certain errors because one cause masks another.

b. Consideration-two: When you trace back through an AND node whose output should be 0, all combinations of inputs leading to a 0 output must be enumerated. However, if one is exploring the situation where one input is 0 and one or more of the others are 1, it is necessary to enumerate all conditions under which the other input can be 1.

c. Consideration-three: When you trace back through an AND node whose output should be 0, only one condition where *all* inputs are zero need be enumerated. (If the AND is in the middle of the graph such that its inputs come from intermediate nodes, there may be an excessive large number of situations under which all of its inputs are 0).

Note: Figure 2.12 on page 55 shows how to apply these considerations.

3. Step-three: Create a column of the decision table for each combination of causes.

4. Step-four: For each combination, determine the states of all other effects and place these in each column in the decision table.

Note that some combinations are ignored; hopefully only high-yield combinations were selected. The final and most difficult step of this approach is to convert the columns of the decision table into test data. This is accomplished in a trial and error manner by inspecting the decision table and generating a test data for each column. For more information regarding the test requirement decision table and the test case derivation, the interested reader should refer to [Myer 79].

2.3 Summary

The D-algorithm and the variations thereof have three essential steps: (1) computation of the embryonic test for the logic block where the fault originates; (2) D-drive which drives forward through the network by sensitizing all paths from the origin of the fault to the primary outputs; and (3) consistency checking, which drives in reverse through the network to the primary inputs, the logic values required to generate the embryonic test and sensitize all paths. At the completion of the consistence operation, the stimuli that detect the assumed fault are generated.

Conceptually, the problem of consistency operation is analogous to the following problem: given a set of logic values at the output of a network, drive them in reverse through the network to find the appropriate inputs. Of all the steps in the D-algorithm, the consistence operation is the most computationally intensive because it is a trial-and-error

procedure that constantly makes decisions to resolve the conflicting assignment of logic values at different gates.

Probabilistic and deterministic test-pattern generations were successfully combined for testing logic boards comprising sequential networks [Agra 72]. However, it is shown in [ShMC 72] that probabilistic test patterns are not always very effective, especially for the class of faults called latent; some latent faults require many more randomly generated attempts to find a test than would be required to completely enumerate all input combinations. Probabilistic test-pattern generation is much more effective on combinational logic than it is on sequential networks, as shown in paper [WiEi 77], where the sequential logic is designed in such a manner as to appear combinational for testing. Combinational logic is the type of logic in which we are primarily interested.

This chapter has given us a basis for our methodology. And we have also described, in detail CCN, a sensitization-based software technique.

Chapter III
INCREMENTAL SENSITIZED TEST CASE DESIGN METHODOLOGY

In Chapter II, we introduced existing path sensitization testing techniques including both hardware and software testing techniques. In this chapter, we describe our approach, an incremental sensitized testing methodology for systematically designing non-redundant high-yield test requirements.

We begin with basic definitions, proofs and examples involving our formalism for representing specifications. This formalism is called a *CCN* (Cause-Effect Combinational Network). Next, we present algorithms for the construction of canonical forms, generation of test requirements based on sensitization of canonical forms, and incremental generation of test requirements by redundancy removal. All algorithms are illustrated using a subset of the Text Reformatter Problem a well-known testing problem. For brevity, this problem is denoted TRP.

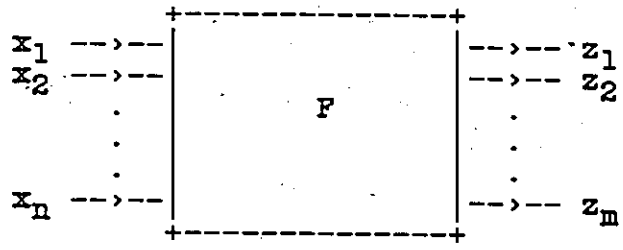
3.1 Combinational Networks

DEFINITION: A *Combinational Network* (abbreviated *CN*) is a feedback-free network that is constructed by means of logical gates {AND, OR, NOT, NAND, NOR} accepting a collection of

"inputs" and generating a collection of "outputs". The output values only depend on the current values of the inputs and are completely independent of previous values of the inputs. Each input and output is *binary-valued*, that is, capable of assuming only two distinct values denoted by $B = \{0,1\}$.

Consider a combinational network with n binary inputs denoted by $x_1, x_2, \ldots, x_n$, and m binary outputs denoted by $z_1, z_2, \ldots, z_m$.

DEFINITION: A *CN*, *N*, is said to *realize a function* $F: B^n \dashrightarrow B^m$ (where $B = \{0,1\}$), if *N* accepts n binary inputs, denoted by $x_1, x_2, \ldots, x_n$, and yields m binary outputs, denoted by $z_1, z_2, \ldots, z_m$, such that $F(x_1, x_2, \ldots, x_n) = (z_1, z_2, \ldots, z_m)$.



$F(x_1, x_2, \ldots, x_n) = (z_1, z_2, \ldots, z_m)$

Figure 3.1: Combinational Network Realizing Combinational Function CF: $B^n \dashrightarrow B^m$

This is illustrated in Figure 3.1. A function which can be represented by a *CN* is called a *Combinational Function*

(abbreviated *CF*). In theory, any *CN*, *N*, (and, hence, the *CF* it realizes), can be specified by means of a truth table, which lists the output values of $z_1, z_2, \ldots, z_m$ for each of the 2^n possible values of the inputs $(x_1, x_2, \ldots, x_n)$ of *N*. In practice, this is not done due to the size of the resulting truth table.

LEMMA 3.1: Given a *CN*, *N*, with *n* inputs and *m* outputs, the combinational function $F: B^n \rightarrow B^m$ realized by this *CN* can be expressed by means of *m single-valued, combinational functions*. Specifically, we can write:

$$F(x_1, x_2, \ldots, x_n) = \{f_1(x_1, x_2, \ldots, x_n),$$
$$f_2(x_1, x_2, \ldots, x_n),$$
$$\vdots$$
$$f_m(x_1, x_2, \ldots, x_n)\},$$

i.e., $f_i: B^n \rightarrow B$ is the *i*th single-valued, combinational function yielding output z_i from inputs $(x_1, x_2, \ldots, x_n)$, such that $z_i = f_i(x_1, x_2, \ldots, x_n)$ where $(i = 1, 2, \ldots, m)$.

PROOF: Essentially, this is merely a formal statement of the observation that each individual output can be expressed in terms of the inputs, according to the subnetwork of all paths from the inputs to that output. Any combinational network can be logically decomposed into a set of combinational subnetworks, one per output [Gill 76, Muro 79, Frie 86].

Q.E.D.

In addition, any *CN* (and therefore *CF*) can be faithfully represented as a *CN* (or *CF*) with only AND, OR, and NOT gates ($\{.,+,'\}$ operations, respectively). Such networks are "standard combinational networks".

DEFINITION: A *standard CN* is a *CN* constructed from only AND, OR and NOT gates.

LEMMA 3.2: Any *CN* with *n* inputs and *m* outputs can be faithfully represented as a *standard CN* with *n* inputs and *m* outputs but possibly with additional gates and links (gate interconnections).

PROOF: This is a standard result which follows from the functional completeness of {AND,NOT} and of {OR,NOT}. Proof that either pair of gates is functionally complete may be found in [Diet 71, Ziss 72, Gill 76] and merely involve replacing NAND and NOR gates by {AND,OR,NOT} gates.

Q.E.D.

DEFINITION: A *standard CF* is a *CF* constructed only with the {.,+,'} operations.

As a corollary of *Lemma 3.2*, and because of the equivalence between *CN's* and *CF's*, we have the following result on *CF's*.

LEMMA 3.3: Any *m-valued CF* with *n* arguments can be faithfully represented as a *standard CF* (but possibly with additional terms).

Henceforth, we confine our attention to only *standard CN's* and *standard CF's*.

3.2 Cause-Effect Combinational Networks Foundations

We now introduce a related type of network that is used by our incremental approach as a vehicle for the test requirement generation methodology. This network can represent a functional specification as a set of logical relationships between "causes" and "effects" (defined below).

DEFINITION: A *cause* (with respect to a specification) is a distinct input condition or an "equivalence class" of input conditions. A cause may be *ternary-valued* (that is, capable of assuming one of three distinct values), which can be either "*Not-Invoked*", "*Invoked*", or "*Unimportant*":

1. A cause is said to be *not-invoked* relative to an effect (output) if its absence is important to the presence or absence of that effect.
2. A cause is said to be *invoked* relative to an effect (output) if its presence is important to the presence or absence of that effect.
3. A cause is said to be *unimportant* relative to the determination of an effect (output) if its value has no impact on the presence or absence of that effect.

   Note: This concept is very similar to the concept of a "don't-care" value [Koha 70, Muro 79].

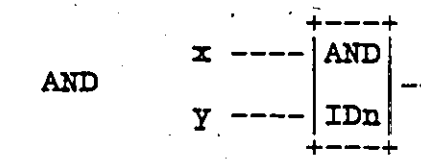
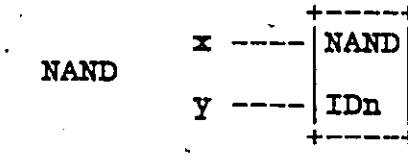
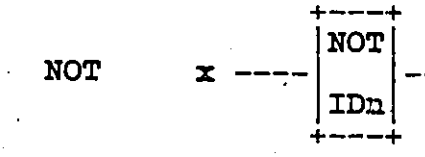
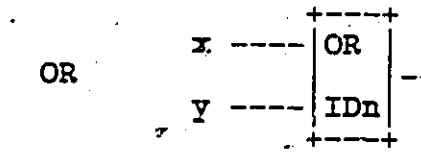
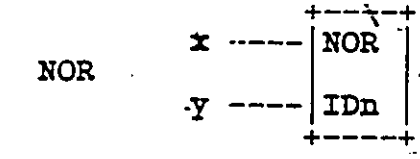
We designate the domain of cause values by $D = \{0, 1, U\}$. The values 0 and 1 correspond exactly to logical "FALSE" and "TRUE" respectively. The value U is defined in terms of logical operations in Table 3.1 on page 67 and in terms of algebraic axioms in Table 3.3 on page 78.

DEFINITION: An *effect* (with respect to a specification) is a distinct output condition or a system transformation. An effect is also ternary-valued, and can be either "*Not-Present*", "*Present*", or "*Unknown*" (the domain of values is *{0,1,U}*). The value of an *effect* is determined according to the logical combination of *causes* as specified in the *CCN* subnetwork defining that *effect*:

1. An effect is said to be *not-present* relative to a set of assignments to inputs (causes) if the specified logical combination of causes in the subnetwork defining that effect yields a logical value of *0*.
2. An effect is said to be *present* relative to a set of assignments to inputs (causes) if the specified logical combination of causes in the subnetwork defining that effect yields a logical value of *1*.
3. An effect is said to be *unknown* relative to a set of assignments to inputs (causes) if the specified assignment of values to these causes in the subnetwork defining that effect is not sufficient to determine a logical value for that effect according to the gate structure of the *CCN* subnetwork.

Truth tables defining AND, OR, NOT, NAND, and NOR gates over ternary valued inputs and outputs are given in Table 3.1 on page 67. The use of the value *U* is motivated by practical testing considerations, wherein only a part of the specification is the focus of interest for each individual test requirement. Refer to Appendix A for a realistic example.

Table 3.1: Cause-Effect Combinational Ternary-Valued Logic Gates.

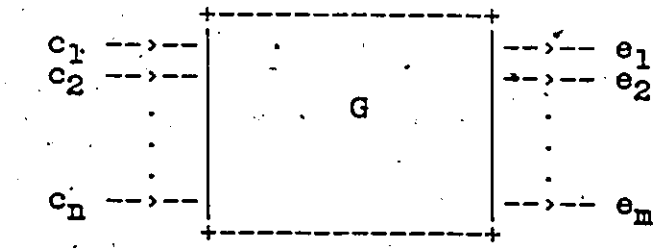
Gate name	Graphic symbol	Algebraic function	Truth table
AND		z = x.y	x y | x.y | (x.y)' 0 0 | 0 | 1 0 1 | 0 | 1 0 U | 0 | 1 1 0 | 0 | 1 1 1 | 1 | 0 1 U | U | U U 0 | 0 | 1 U 1 | U | U U U | U | U
NAND		z = (x.y)'	
NOT		z = x'	x | x' 0 | 1 1 | 0 U | U
OR		z = x+y	x y | x+y | (x+y)' 0 0 | 0 | 1 0 1 | 1 | 0 0 U | U | U 1 0 | 1 | 0 1 1 | 1 | 0 1 U | 1 | 0 U 0 | U | U U 1 | 1 | 0 U U | U | U
NOR		z = (x+y)'	

Each gate has <u>one</u> or <u>two</u> *ternary-valued* (input) variables designated by *x* and *y* and <u>one</u> *ternary-valued* (output) variable designated by *z*. Inside each gate there are two labeling fields. The lower labeling field is marked with a 3-digit identification number. Its purpose is to identify that gate in the given *CCN*. The upper labeling field is marked with the logical type of the gate {AND, OR, NOT, NAND, NOR}.

Now we can formally define a cause-effect combinational network *CCN*. Let us consider a network with *n ternary-valued causes* denoted by $c_1, c_2, \ldots, c_n$, and *m ternary-valued effects* denoted by $e_1, e_2, \ldots, e_m$.

DEFINITION: A *Cause-Effect Combinational Network* (abbreviated *CCN*) is a *CN* whose *effect* values depend only on the current values of the *causes* and are completely independent of previous values of the *causes*.

DEFINITION: A *CCN*, *M*, is said to *realize a function* $G: D^n \dashrightarrow D^m$ (where $D = \{0,1,U\}$), if *M* accepts *n ternary-valued causes*, denoted by $c_1, c_2, \ldots, c_n$, and yields *m ternary-valued effects*, denoted by $e_1, e_2, \ldots, e_m$, such that $G(c_1, c_2, \ldots, c_n) = (e_1, e_2, \ldots, e_m)$. This is illustrated in Figure 3.2. A function which can be represented by a *CCN* is called a *Cause-Effect Combinational Function* (abbreviated *CCF*).



$G(c_1, c_2, \ldots, c_n) = (e_1, e_2, \ldots, e_m)$

Figure 3.2: Cause-Effect Combinational Network Realizing the CCF $G: D^n \dashrightarrow D^m$

In theory, any *CCN*, *M*, (and, hence, any *CCF* it realizes), can be specified by means of a ternary truth table, which lists the effect values of $e_1, e_2, \ldots, e_m$ for each of the 3^n possible assignment values to the causes $\{c_1, c_2, \ldots, c_n\}$ of *M*.

LEMMA 3.4: Given a *CCN*, *M*, with *n* causes and *m* effects, the cause-effect combinational function $G: D^n \rightarrow D^m$ realized by this *CCN* can be expressed by means of *m single-valued*, cause-effect combinational functions. Specifically, we can write:

$$G(c_1, c_2, \ldots, c_n) = \{g_1(c_1, c_2, \ldots, c_n), g_2(c_1, c_2, \ldots, c_n), \ldots, g_m(c_1, c_2, \ldots, c_n)\},$$

Where $g_i: D^n \rightarrow D$ is the *ith* single-valued, cause-effect combinational function yielding effect e_i from causes $(c_1, c_2, \ldots, c_n)$. Accordingly, for convenience, we may write $G: D^n \rightarrow D^m$ as $G = \{g_1, g_2, \ldots, g_m\}$ where each $g_i: D^n \rightarrow D$.

PROOF: This *Lemma* simply states that any cause-effect combinational function can be decomposed into a set of single-valued cause-effect combinational functions, one per effect. The *Lemma* follows by considering each *CCN* subnetwork corresponding to one effect, one at a time.

Q.E.D.

In addition, we will now show that any *CCN* (and therefore, any *CCF*) can be faithfully represented as a *CCN* (or *CCF*) involving only AND, OR, and NOT gates (i.e., $\{.,+,'\}$ operations, respectively). Such networks are denoted "standard cause-effect combinational networks".

DEFINITION: A *standard CCN* is a *CCN* constructed only from AND, OR, and NOT gates.

LEMMA 3.5: Given a *CCN*, *M*, with at least one NAND, or NOR gate, there exists an equivalent *CCN*, *M'*, with one less NAND or NOR gate (but at the additional cost of a gate).

PROOF: Without loss of generality, consider any NAND gate (a similar argument holds for NOR gates). Rewrite that NAND (or NOR) gate in terms of OR-(or AND) and NOT gates as shown in Figure 3.3 on page 71. By the *ternary-valued* tables in Figure 3.3 on page 71, we can demonstrate that the NOT-OR (or NOT-AND) combinations are functionally identical to NAND (or NOR) respectively.

So this construction yields a new, equivalent *CCN* *M'*, with one-fewer NAND (or NOR) gate (but at the expense of additional AND, OR, or NOT gates).

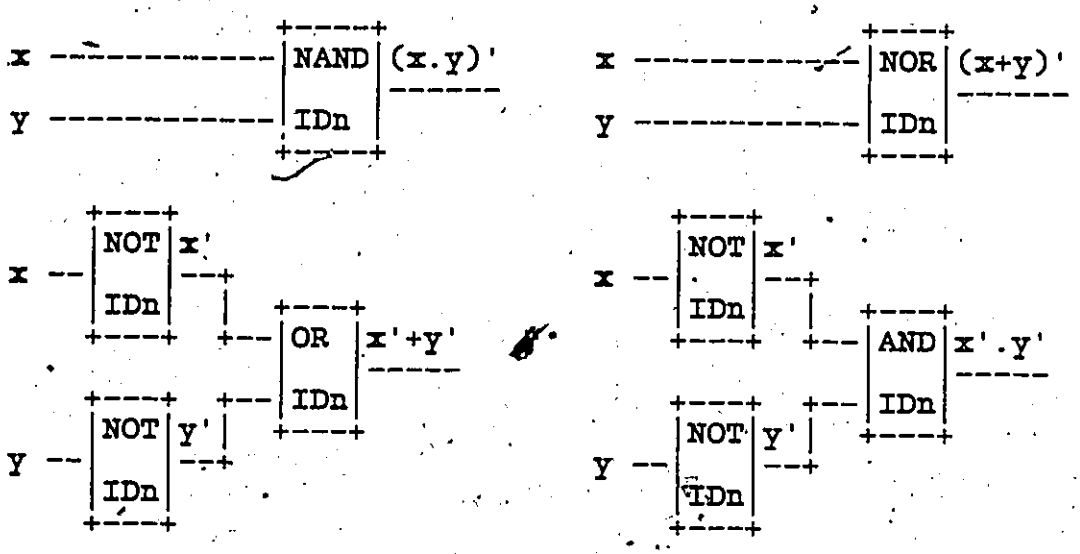
Q.E.D.

Now, using *Lemma 3.5*, we are able to prove:

LEMMA 3.6: Any *CCN* with *n* causes and *m* effects can be faithfully represented as a *standard CCN* with *n* causes and *m* effects but possibly with additional gates and links (gate interconnections).

PROOF: The argument involves induction on the number of NAND and NOR gates.

BASIS: If the CCN has only 1 NAND and NOR gate, then apply *Lemma 3.5*.



(x.y)'	x'+y'
1	1
1	1
1	1
1	1
0	0
U	U
1	1
U	U
U	U

x	y	x'	y'	x.y	x+y
0	0	1	1	0	0
0	1	1	0	0	1
0	U	1	U	0	U
1	0	0	1	0	1
1	1	0	0	1	1
1	U	0	U	U	1
U	0	U	1	0	U
U	1	U	0	U	1
U	U	U	U	U	U

(x+y)'	x'.y'
1	1
0	0
U	U
0	0
0	0
0	0
U	U
0	0
U	U

Figure 3.3: NAND and NOR Representation with NOT-OR and NOT-AND combinations.

INDUCTION HYPOTHESIS: Assume *CCN's* containing *k* or fewer NAND or NOR gates can be faithfully represented by *standard CCN's*.

Given a *CCN* with *k+1* NAND or NOR gates, apply *Lemma 3.5* to reduce the number of such gates by 1. By induction, the result holds.

Q.E.D.

DEFINITION: A *standard CCF* is a *CCF* constructed only with {., +, and '} operations.

As a corollary of *Lemma 3.6*, and because of the equivalence between *CCN's* and *CCF's*, we have the following result.

LEMMA 3.7: Any m valued *CCF* with n arguments can be represented faithfully as a *standard CCF* (but possibly with additional terms).

Henceforth, we confine our attention to only *standard CCN's* and *standard CCF's*. For brevity, we will drop the term *standard* and call them simply *CCN's* and *CCF's*.

We have now completed our discussion covering the fundamentals of *CN's*, *CF's*, *CCN's*, and *CCF's* as well as their *standard* constructs. In the next two sections, we give an illustration of the use of *CCN's*, as well as guidelines for constructing a *CCN* from natural language specifications.

3.2.1 An Example CCN

Until now we have limited our discussion to various *Definitions* and *Lemmas* on *standard CCN's*. Here, we illustrate the concepts by an example *standard CCN* based on Walsh's formulation of the TRP [Wals 83].

This problem is of particular interest to researchers in software testing and represents somewhat of a "benchmark" problem for assessing testing techniques. Goodenough and Gerhart have used this problem to illustrate their theoretical approach towards test data selection [GoGe 75]. Probert and Ural, as well

as a number of other researchers including Walsh, have used this problem as a useful example [PrUr 84]. However, the work by Walsh is closest to our approach. Accordingly we adopt his formalization of the problem [Wals 83].

For now, we consider only a part of TRP whose *causes*, *intermediate cause*, and *effects* are outlined in Table 3.2 on page 74. *Intermediate causes* denote the set of conditions which are defined by internal gates in the network. This is illustrated in our example *CCN's* shown in Figure 3.4 on page 75. Information on the interpretation and semantics of causes ; intermediate causes and effects will be provided later. Note that the graph in Figure 3.4 on page 75 is indeed a *standard CCN* realizing the *standard CCF's* shown in Figure 3.5 on page 76.

To represent a cause entry point in a *standard CCN* (i.e., Figure 3.4 on page 75), we use a "pseudo-gate" where this entry point is stored. We call this pseudo-gate a *buffer* (abbreviated BUF). This BUF gate does not do any logic operation. It always has one input (the cause entry point), and can have one, two, or more than two outputs ("duplications" of its cause entry point). Therefore, any cause needed in the network is drawn from the output of a BUF gate which carries a duplication of this cause.

Some of the gates in Figure 3.4 on page 75, other than NOT's and BUF's can accept more than two causes, or intermediate causes. A gate in a standard CCN can have multiple inputs if the ternary operation it represents is *commutative* and *associative*. In particular, our ternary-valued AND and OR gates possess these

Table 3.2: Definition of Causes, Intermediate Causes, and Effects for a subset of the TRP.

6 Causes: c_k, (k = 1,2, ... ,6)

c_1 : Character is NL
c_2 : Character is BL
c_3 : Word found will fit on current line
c_4 : At least one character found and not printed
c_5 : A word was already printed on the current line
c_6 : Character is ET

4 Intermediate causes: i_j, (j = 1,2, ... ,4)

i_1 : Requires one and only one of c_1, or c_2, or c_6
i_2 : Both conditions defined by i_1 and c_4 hold
i_3 : Both causes c_3 and c_5 hold
i_4 : Condition defined by i_3 does not hold

2 Effects: e_i, (i = 1,2)

e_1 : (BL and work area for storing a word) are printed
e_2 : (NL and work area for storing a word) are printed

two properties (Table 3.3 on page 78). Therefore, AND and OR gates can have three or more inputs (e.g., gate (i_1,OR) of Figure 3.4 on page 75). Clearly, the corresponding standard CCF to the *standard CCN* (Figure 3.4 on page 75), is $G: D^6 \rightarrow D^2$. The expanded form $G = \{g_1, g_2\}$ is shown in Figure 3.5 on page 76. The rules for deriving these *CCF's* are presented in a later section.

Thus, the *CCN* in Figure 3.4 on page 75 is best viewed as being logically partitioned into three areas: *causes, intermediate causes* and *effects*. Each such area has its own gate types, namely {BUF}, {AND,OR,NOT}, and {AND}, respectively. Causes are denoted by c_k (where, $k = 1,2, \ldots n$), intermediate causes by i_j (where, $j = 1,2, \ldots ,r$), and effects by e_i (where, $i = 1,2, \ldots m$).

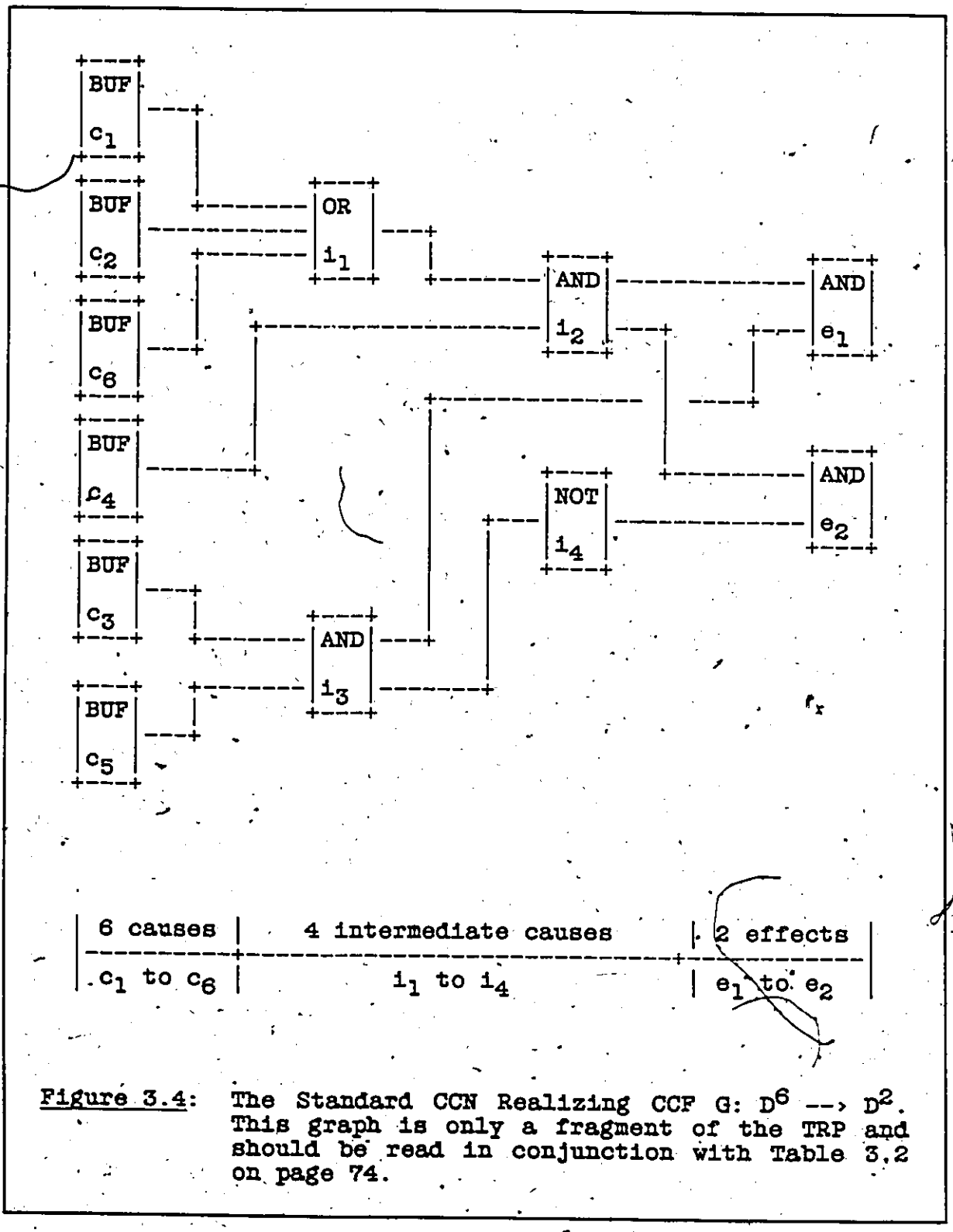


Figure 3.4: The Standard CCN Realizing CCF G: $D^6 \dashrightarrow D^2$. This graph is only a fragment of the TRP and should be read in conjunction with Table 3.2 on page 74.

$$g_1 = i_2 \cdot i_3$$
$$= c_4 \cdot i_1 \cdot c_3 \cdot c_5$$
$$= c_4 \cdot (c_1 + c_2 + c_6) \cdot c_3 \cdot c_5$$

$$g_2 = i_2 \cdot i_4$$
$$= c_4 \cdot i_1 \cdot i_3'$$
$$= c_4 \cdot (c_1 + c_2 + c_6) \cdot (c_3 \cdot c_5)'$$
$$= c_4 \cdot (c_1 + c_2 + c_6) \cdot (c_3' + c_5')$$

Figure 3.5: An Expanded Representation of $G = \{g_1, g_2\}$. Where $G = \{g_1, g_2\}$: $D^6 \rightarrow D^2$ is a *standard CCF* realized by the *Standard CCN* of Figure 3.4 on page 75.

3.2.2 Guidelines for Constructing a CCN from Natural Language Specifications

This section addresses the transformation of functional specifications into a *standard CCN*. The transformation of this *standard CCN* into its *standard CCF* is an automated step in our approach and is presented in a later section.

We adopt a similar approach to Elmendorf and Myers [Elme 75, Myer 79]. The procedure for generating a *standard CCN* is as follows:

Procedure:

1. List *vertically* all *causes* on the *left side* of a sheet of paper and similarly, all *effects* on the *right side*.

2. Link causes to effects with a structure of logical relationships, each one representing a conditional statement (IF...THEN...) found in or implied by the specification. The logical relationship between causes and effects that represents an effect which, in turn, becomes a cause is called an *intermediate cause*.
3. If there is only one cause on the antecedent side and one effect on the consequent side of a relationship, the cause is simply connected to the effect.
4. Multiple *causes*, multiple *intermediate causes* and multiple *effects* in a relationship are represented by *fanin* and *fanout* on the corresponding gate interconnections.
5. Label the gate fanin relationships according to the connectives appearing in the antecedents of the conditional statements. The eligible connectives are {AND,OR,NOT}.

Recall that *CCN's* express only combinational relationships between causes and effects. Permutations of causes, time delays and feedback from effect to causes cannot be graphed. Fortunately, in functional specifications, such complications are the exception rather than the rule. When they do occur, they must either be buried within the definitions of individual gates or be implicitly understood.

Typically, the *CCN* construction process is highly subjective and uses iteration to obtain a complete and consistent representation of the specification. In the next chapter, we describe a tool developed to support this initial *standard CCN* derivation step.

3.2.3 Cause-Effect Combinational Algebra

The mathematical system of binary logic is also known as Boolean algebra, or switching algebra [Diet 71, Gill 76]. This algebra is used to describe the operation of complex networks of digital circuits. Designers of digital systems use Boolean algebra to transform logic networks to algebraic expressions and vice versa [Muro 79, Goth 82].

In our approach, a "cause-effect combinational algebra" underlies the analysis and synthesis of *standard CCN's*, as well as the evaluation and conversion of *CCF's* from one form to another.

DEFINITION: A *cause-effect combinational algebra*, is an algebra denoted by $[D; \{.,+,'\}]$ (where D is the *ternary* domain $\{0,1,U\}$), satisfying the axioms given in Table 3.3.

Table 3.3: Cause-Effect Combinational Algebra Axioms.

Name of Axiom	Over "+" or OR	Over "." or AND
Commutative	$x+y = y+x$	$x.y = y.x$
Associative	$x+(y+z) = (x+y)+z$	$x.(y.z) = (x.y).z$
Distributive	$x.(y+z) = x.y + x.z$	$\overline{x.(y.z)} = \overline{(x.y).z}$
Identity	$x+0 = 0+x = x$	$x.1 = 1.x = x$
Involution	$x'' = x$	$\overline{\overline{x}} = x$
Idempotent	$x + x = x$	$x.x = x$
Null	$x + 1 = 1$	$x.0 = 0$
De Morgan's	$(x+y)' = x'.y'$	$(x.y)' = x' + y'$
U-Complement	$U' = U$	$\overline{U} = U$
U-Identity	$U+0 = 0+U = U$	$U.1 = 1.U = U$
U-Idempotent	$U + U = U$	$U.U = U$
U-Null	$U+1 = 1+U = 1$	$U.0 = 0.U = 0$

LEMMA 3.8: The above cause-effect combinational algebra $[D; \{.,+,'\}]$ is *homomorphic* to the *standard switching (Boolean) algebra* $[B; \{.,+,'\}]$.

PROOF: This result follows by noting that all algebraic laws over $B = \{0,1\}$ are also satisfied over $D = \{0,1,U\}$.

Q.E.D.

Thus, operations over the *ternary* alphabet related in a natural way to normal *binary* operations. In the next section, we introduce the canonical forms of the *CCF* of a *CCN* and illustrate their construction through examples.

3.3 Canonical Forms of the CCF of a CCN

One important result we can carry over from switching algebra to cause-effect combinational algebra is that concerning minterm and maxterm normal forms for *CCF's* [Gill 76]. We now define two corresponding canonical forms for any *CCF*.

3.3.1 Introduction

DEFINITION: A *Cause-Effect Literal*, is a cause variable $\{c_1, c_2, \ldots, c_n\}$ or its complement. Note that all *variables*, *literals*, etc., are henceforth defined over the *ternary* domain $D = \{0,1,U\}$.

DEFINITION: A term s is called a *k-ary sum-term* if s is a "disjunction" of k distinct *literals* (i.e., $s = w_1+w_2+, \ldots, +w_k$ where w_i is a *literal*).

DEFINITION: A term p is called a *k-ary product-term* if p is a "conjunction" of k distinct literals (i.e., $p = w_1.w_2., \ldots, .w_k$ where w_i is a *literal*).

DEFINITION: The number of literals in a sum-term s or a product-term p is called the *cardinality* of s or p and is denoted by $|s|$ or $|p|$ respectively.

The two canonical forms for any *CCF* of the m effects $G(c_1, c_2, \ldots, c_n) = (e_1, e_2, \ldots, e_m)$ realized by a *CCN* are now defined.

3.3.2 Levels of a Single-Effect Standard CCN

It is now necessary to define a property of a *single-effect standard CCN* stemming from its gate structure. We now define the *Level* of a *single-effect standard CCN* to which a specific gate A {AND,OR,NOT} belongs.

DEFINITION: Given a gate A {AND,OR,NOT} and an effect e (i.e., *single-effect standard CCN*), the *level* of gate A with respect to effect e is denoted by $L_e(A)$, and is defined as follows:

- The *level* of a gate A at effect e is set to 0, and is denoted by $L_e(A) = 0$. Note that the gate A at *level* 0 must always be the AND gate. Thus, $L_e(e,AND) = 0$. This is a requirement of the *single-effect standard CCN*.
- The *level* of gate A with respect to effect e is set to 1 and is denoted by $L_e(A) = 1$, if an output from gate A goes directly to effect e (i.e., becomes input to gate (e,AND) at *level* $L_e(e,\text{AND}) = 0$).
- Let $\{B_1, B_2, \ldots, B_k\}$ be gates B_i {AND,OR,NOT} with respect to effect e connected via an output (i.e., fanout) of gate A. Then, $L_e(A)$ is set to $\underline{\min}\{L_e(B_i)\}+1$, and is denoted by $L_e(A) = \underline{\min}\{L_e(B_i)\}+1$, where $i = 1,2, \ldots, k$.

The *level* of a gate A is always defined with respect to its effect e and structure of its *single-effect standard CCN*.

EXAMPLE: For example, the *level* of gate (i_3, AND) with respect to effects e_1 and e_2 of the *two-effect standard CCN* in Figure 3.4 on page 75, is $L_{e_1}(i_3, \text{AND}) \neq L_{e_2}(i_3, \text{AND})$.

This is because the *level* due to gate (i_3, AND) with respect to effect e_1 is set to *1*, while with respect to effect e_2 is set to *2*. In turn, the reason for this is because the number of *levels* along the path with respect to effect e_1 of gate (e_1, AND) leading to target gate (i_3, AND) is <u>two</u>: $L_{e_1}(e_1, \text{AND}) = 0$, followed be $L_{e_1}(i_3, \text{AND}) = 1$.

On the other hand, the number of *levels* along the path with respect to effect e_2 from gate (e_2, AND) leading to the same target gate (i_3, AND) is <u>three</u>: $L_{e_2}(e_2, \text{AND}) = 0$ followed by $L_{e_2}(i_4, \text{NOT}) = 1$ and finally by $L_{e_2}(i_3, \text{AND}) = 2$.

3.3.3 PoS Canonical Form

<u>DEFINITION</u>: The *<u>Product of Sum-terms Canonical Form</u>* (abbreviated *PoS*), is a "minimal" representation of a *single-valued standard CCF* (i.e., $e = g(c_1, c_2, \ldots, c_n)$) as a *product* of *sum-terms* in the form $e = s_1 . s_2 . , \ldots , . s_k$ where each *sum-term* s_1 contains only distinct *literals*.

Note that "minimal" implies that a cause and its complement will not both be present in the same *sum-term*.

3.3.3.1 An Example of PoS Canonical Form

Let us illustrate the *PoS canonical form* of a *single-effect standard CCN*. Consider *effect* e_1 of the *two-effect standard CCN* of Figure 3.4 on page 75. We now illustrate how to obtain its *PoS canonical form* (abbreviated *PoS*(g_1)). The general mechanism for doing so is described in *Algorithm 3.1*.

Briefly, *PoS*(g_1) is obtained by tracing all levels (Section 3.3.2) of gates starting from effect e_1 (Figure 3.4 on page 75), until only *literals* appear in the resulting *single-valued standard CCF*.

$$\begin{aligned} PoS(g_1) &= i_2 \cdot i_3 \\ &= c_4 \cdot i_1 \cdot c_3 \cdot c_5 \\ &= c_4 \cdot (c_1 + c_2 + c_6) \cdot c_3 \cdot c_5 \end{aligned}$$

Similarly,

$$PoS(g_2) = c_4 \cdot (c_1 + c_2 + c_6) \cdot (c_3' + c_5')$$

Figure 3.6: PoS(g_1) and PoS(g_2) Standard CCF's Canonical Forms. The above canonical forms are realized by the standard CCN of Figure 3.4 on page 75.

Starting from effect e_1 (i.e., gate (e_1,AND), Figure 3.4 on page 75), the effect e_1 holds true if both (due to the AND operation, Table 3.1 on page 67) conditions i_2 and i_3 (*intermediate causes*, Table 3.2 on page 74) hold true as well. This is represented by the first line of *PoS*(g_1) in Figure 3.6, which actually marks the end of the *first level* of *PoS*(g_1). Continuing with Figure 3.4 on

page 75, we consider i_2 and i_3 separately. First, in gate (i_2,AND), the intermediate cause i_2 holds true if both cause c_4 and intermediate cause i_1 hold true. On the other hand, in gate (i_3,AND), intermediate cause i_3 holds if both causes c_3 and c_5 hold true (see Table 3.2 on page 74). The above is represented by the second line $PoS(g_1)$ in Figure 3.6 on page 82, which also marks the end of the *second level* of the $PoS(g_1)$.

At a last step, in gate (i_1,OR), the intermediate cause i_1 holds true, if and only if at least one of its three causes c_1, c_2, or c_6 (Table 3.2 on page 74), holds true.

After this last substitution, that is, expressing the intermediate cause i_1 in terms of its causes $\{c_1, c_2, c_6\}$, the $PoS(g_1)$ is finally obtained; It is represented in the third line of $PoS(g_1)$ in Figure 3.6 on page 82. This also marks the end of the *third* (last) *level* of $PoS(g_1)$. Similarly, we can derive the *PoS* of g_2, shown also in Figure 3.6 on page 82.

Note that the canonical form $PoS(g_1)$ is the *product* (AND) of four sum-terms, which we denote by (s_1, s_2, s_3, s_4) of cardinality $|s_1| = 1$, $|s_2| = 3$, $|s_3| = 1$, and $|s_4| = 1$ each.

3.3.4 SoP Canonical Form

DEFINITION: The *Sum of Product-terms Canonical Form* (abbreviated *SoP*), is a "minimal" representation of a *single-valued standard CCF* (i.e., $e = g(c_1, c_2, \ldots, c_n)$) as a *sum* of *product-terms* in the form $e = p_1 + p_2 + \ldots + p_k$ where each product-term p_i contains only distinct literals.

Note that "minimal" implies that a cause and its complement will not both be present in the same *product-term*.

3.3.4.1 An Example of SoP Canonical Form

To illustrate the *SoP canonical form* of a *single-effect standard CCN*, consider effect e_1 of the *CCN* in Figure 3.4 on page 75. Its *SoP* canonical form (abbreviated $SoP(g_1)$), is derived from $PoS(g_1)$ in Figure 3.6 on page 82. The general mechanism for doing so is described in *Algorithm 3.2*. $SoP(g_1)$ is given in Figure 3.7.

$$SoP(g_1) = c_4 \cdot c_3 \cdot c_5 \cdot c_1 + c_4 \cdot c_3 \cdot c_5 \cdot c_2 + c_4 \cdot c_3 \cdot c_5 \cdot c_6$$

Similarly,

$$SoP(g_2) = c_4 \cdot c_1 \cdot c_3' + c_4 \cdot c_1 \cdot c_5' + c_4 \cdot c_2 \cdot c_3' + c_4 \cdot c_2 \cdot c_5' + c_4 \cdot c_6 \cdot c_3' + c_4 \cdot c_6 \cdot c_5'$$

Figure 3.7: $SoP(g_1)$ and $SoP(g_2)$ Canonical Forms. Derived from $PoS(g_1)$ and $PoS(g_2)$ in Figure 3.6 on page 82.

Clearly, from $SoP(g_1)$ in Figure 3.7, for effect e_1 to hold, it is sufficient that one of its product-terms be true. Note that, $SoP(g_1)$ is the *sum* (OR) of three product-terms (p_1, p_2, p_3) of

cardinality $|p_1| = 4$, $|p_2| = 4$, and $|p_3| = 4$. Similarly, *SoP*(g_2), derived from *PoS*(g_2), is shown in Figure 3.7 on page 84.

3.4 Algorithmic Construction of Canonical Forms

THEOREM 3.1: Any well-formed formula involving {.,+,'} operations, $\{c_1, c_2, \ldots, c_n\}$ variables and parentheses can be represented in *SoP* or *PoS* canonical form.

PROOF: The demonstration of this fact over the domain $B = \{0,1\}$ can be found in most standard references for propositional logic [Klee 67]. By induction on the size of the formula, we can show this theorem also holds over the *ternary* domain $D = \{0,1,U\}$ based on the *cause-effect combinational algebra axioms* (stated in Table 3.3 on page 78).

Q.E.D.

COROLLARY 3.1: Every *single-valued CCF* can be represented in *PoS* or *SoP* canonical form.

PROOF: The proof follows from *Theorem 3.1* and *Lemma 3.7*. Therefore, the *CCF* of any *CCN* can be represented as a family of *SoP* or *PoS* canonical forms.

Q.E.D.

Some types of redundancy arises in the construction of the canonical form. To eliminate obvious redundancy, we make the following assumption:

Fundamental assumption: For the logical integrity of specifications no two distinct effects from an *m-valued CCF*

(i.e., e_i and e_j) may have an identical canonical form: $PoS(g_i) = PoS(g_j)$, and $SoP(g_i) = SoP(g_j)$. This usually indicates a design or specification error which can be masked by the complexity of the *CCN*.

Logical redundancy: The construction of *SoP* and *PoS* canonical forms (discussed next), deal with a type of information redundancy, called *logical redundancy*. Logical redundancies involve superfluous relationships between literals in sum-terms and product-terms. These redundancies are detected and removed as soon as they are derived by applying the "involution" and "idempotent" axioms in Table 3.3 on page 78. Thus, the resulting *PoS* and *SoP* canonical forms for a *single-valued CCF* contain no logical redundancies.

Given a *standard CCN* (e.g., Figure 3.2 on page 68), the *representation* problem is to obtain its *CCF's* (i.e., $G = \{g_1, g_2, \ldots, g_m\}$: $D^n \dashrightarrow D^m$). This task is carried out in our approach by considering one *single-effect CCF* (i.e., g_i) at a time.

Thus, without loss of generality in this section, we confine our attention to single-effect CCN's.

3.4.1 Construction of PoS Canonical Form

Armstrong, has proposed a method for deriving a *nearly* minimal set of fault-detection tests, making use of the *Equivalent Normal Form* (see Sections 2.1.2.5 and 2.1.3.1) of the circuit [Arms 66]. Similarly, in our approach to obtain the *PoS* canonical form of a *single-effect CCN* we need a formula which help us to represent the *output* of each gate as a function of its *inputs*.

Applying this gate representation to all gates {AND,OR,NOT} in the *single-effect CCN*, we obtain an expression (usually in a factored form - *PoS* canonical form) of the effect e in terms of its causes $\{c_1, c_2, \ldots, c_n\}$. The general mechanism for doing so is described in *Algorithm 3.1*. However, the gate {AND,OR,NOT} representation is defined as follows:

DEFINITION: The *Gate Traversal Form* (abbreviated *GTF*) of a *standard CCN* effect gate is a formula obtained by recursively expressing the *output* of each gate in the *subnetwork* defined by that effect as a function of its *inputs*. If the inputs to a gate are denoted by $\{x,y\}$ and the output by $\{z\}$ then:

1. $GTF_{AND}(z) = GTF(x) \cdot GTF(y)$
2. $GTF_{OR}(z) = (GTF(x) + GTF(y))$
3. $GTF_{NOT}(z) = (GTF(x))'$

We now give an iterative algorithm to construct the *PoS* form of a *CCN* based on this definition and on a level-by-level construction. *Levels* were defined in Section 3.3.2.

ALGORITHM 3.1: *Construction of PoS Canonical Form*

Purpose:

To derive the *PoS canonical form* of the corresponding *single-valued standard CCF*, $e = g(c_1, c_2, \ldots, c_n)$ given a *single-effect CCN*, M, with h *levels* of gates $L_1 < L_2 < \ldots < L_h$.

Procedure:

1. At *level 1* (denoted by L_1) set the *PoS* intermediate representation to the *GTF(e)*. Call this PoS_1.

   Note: L_1 always contains one gate. In our experience with realistic *CCN's*, we have found that this gate is always an AND. Therefore, we make this assumption in this and all subsequent algorithms.

2. Proceed to the next-level, denoted by L_2. This level may contain any number of {AND,OR,NOT} gates.

   a. Translate every gate at level L_t into its *GTF*.

   b. Substitute all *GTF's* of step (2.a.) into the intermediate representation PoS_{t-1} to yield the current PoS_t.

   c. Use "De Morgan's" axioms (Table 3.3 on page 78) in PoS_t to remove parentheses where applicable.

   d. Finally, at L_t, apply *logical redundancy* removal (Section 3.4) to *literals* and *sum-terms*.

3. Apply step (2.) repeatedly to the remaining $(h-t)$ *levels*, until L_h is reached. The resulting PoS_h is expressed only in terms of *literals*. Thus, the procedure is terminated.

The correctness of this algorithm follows from the fact that its *procedure* always terminates and the corresponding PoS_h is expressed only in terms of *products-of-sums of literals*. Thus, *PoS* form is the *CCN* equivalent of *conjunctive-normal-form* [Koha 70, Muro 79, Mano 84].

3.4.1.1 An Example

We now illustrate *Algorithm 3.1* ("*Construction of PoS Canonical Form*") by applying it to a *single-effect subnetwork* of the *CCN* of Figure 3.4 on page 75. Specifically, we consider effect e_2, and will derive its *PoS* canonical form, $PoS(g_2)$. Following the steps of *Algorithm 3.1* applied to e_2 we have:

1. Step 1: At L_1, we have the *single-effect* e_2 which is an AND gate (e_2, AND). $GTF_{AND}(e_2) = i_2.i_4$, thus, our *PoS* intermediate form for *level 1* is:

   $$PoS_1(e_2) = i_2.i_4 \qquad - L_1$$

2. Step 2: At L_2, we have two gates (i_2, AND) and (i_4, NOT):
   a. $GTF_{AND}(i_2) = c_4.\overline{i_1}$ and $GTF_{NOT}(i_4) = i_3'$.
   b. Substituting these *GTF's* in $PoS_1(e_2)$ the *PoS* intermediate form at L_2 is:

      $$PoS_2(e_2) = c_4.i_1.i_3' \qquad - L_2$$

   c. "De Morgan's" axioms (Table 3.3 on page 78) do not apply. As well no *logical redundancies* are detected (we have ~~three~~ distinct sum-terms, each of cardinality 1). Thus, our current *PoS* intermediate form remains the same:

      $$PoS_2(e_2) = c_4.i_1.i_3' \qquad - L_2$$

3. Step 3: The "termination condition" of the algorithm does not hold, since the current $PoS_2(g_2)$ still contains the intermediate terms: $\{i_1, i_3'\}$.

4. Step 2: We continue with the next level. L_3 contains two gates (i_1,OR) and (i_3,AND):
   a. $GTF_{OR}(i_1) = (c_1+c_2+c_6)$ and $GTF_{AND}(i_3) = c_3.c_5$.
   b. Substituting the *GTF's* in $PoS_2(e_2)$ the *PoS* intermediate form at L_3 is:

      $$PoS_3(e_2) = c_4.(c_1+c_2+c_6).(c_3.c_5)' \quad - L_3$$

   c. We now apply "De Morgan's" axiom. Therefore:

      $$PoS_3(e_2) = c_4.(c_1+c_2+c_6).(c_3'+c_5') \quad - L_3$$

   d. No *logical redundancies* are detected. Therefore, our *PoS* intermediate form remains as above.
5. Step 3: The "termination condition" of the algorithm holds. $PoS_3(e_2)$, is expressed only in terms of *literals* and therefore we terminate the process.
6. Step 4: The resulting $PoS_3(e_2)$ is the *product* of three *sum-terms*.

This sequence of steps is given in more compact form in Figure 3.8 on page 91.

$PoS(e_2) = i_2 \cdot i_4$ - L_1

$= c_4 \cdot i_1 \cdot i_3'$ - L_2

$= c_4 \cdot (c_1 + c_2 + c_6) \cdot (c_3 : c_5)'$ - L_3

MSG De Morgan's Axiom on: $(c_3 \cdot c_5)'$
$(x \cdot y)' \rightarrow x' + y'$

$= c_4 \cdot (c_1 + c_2 + c_6) \cdot (c_3' + c_5')$ - L_3

s_1 s_2 s_3 (sum-terms)

Figure 3.8: An Example Illustrating the Algorithmic Construction of the $PoS(e_2)$ Canonical Form. This form is based on the *CCN* of Figure 3.4 on page 75.

3.4.2 Construction of SoP Canonical Form

The *SoP* form can be obtained directly by multiplying out the *PoS* form according to the "distribution" axiom of "." over "+" shown in Table 3.3 on page 78. The steps of this multiplication are gathered into an algorithm below.

ALGORITHM 3.2: *Construction of SoP Canonical Form from PoS Canonical Form*

Purpose:

Given a *PoS* canonical form of an effect *e* in a *CCN*, namely *PoS(e)* = $s_1 \cdot s_2 \cdot , \ldots , \cdot s_u$, construct its corresponding *SoP* canonical form.

Procedure:

1. Scan through (left-to-right) all u sum-terms of the $PoS(e)$ canonical form:

   a. Gather together all sum-terms of cardinality equal to 1 in a combined product. Call this $PoS_{part1}(e)$.

      Note: This product may contain no terms. In this case, we set $PoS_{part1}(e)$ to "TRUE".

   b. The remaining *complex* sum-terms (those with cardinality higher than 1) are concatenated into $PoS_{part2}(e)$.

   c. Concatenate the above $PoS_{part1}(e)$ and $PoS_{part2}(e)$ into a $PoS_{new}(e)$. Note that $PoS_{new}(e)$ is logically equivalent to $PoS(e)$ because of the "commutativity" and "associativity" of "." (Table 3.3 on page 78).

2. Multiply left-to-right all *complex* sum-terms by considering two sum-terms at a time as the following algorithm indicates, until all *complex* sum-terms have been considered:

   Let:

   $$SoP_{temp}(e) = PoS_{part1}(e)$$

   For all *complex* sum-terms s_1 do:

   $$SoP_{temp}(e) = SoP_{temp}(e) \cdot s_1$$

   Note that the resulting $SoP_{temp}(e)$ is the sum of v product-terms where $v = |s_1| \times |s_2| \times \ldots \times |s_u|$. Thus, we have

   $$SoP_{temp}(e) = p_1 + p_2 + \ldots + p_v$$

3. Apply *logical redundancy* removal to $SoP_{temp}(e)$ (Section 3.4).

4. The resulting $SoP_{temp}(e)$ redundancy-free expression is the unique *SoP* canonical form.

3.4.2.1 A Special Case

A special case occurs when <u>every</u> sum-term s_i in $PoS(e) = s_1.s_2. \ldots .s_u$ has cardinality $|s_i| = 1$ (i.e., <u>each</u> sum-term involves a <u>single</u> *literal*).

In this case, $v = |s_1| \times |s_2| \times \ldots \times |s_u| = 1$. Therefore, we have <u>one</u> product-term and the *SoP* canonical form will be $SoP(e) = p_1$, where $p_1 = s_1.s_2. \ldots .s_u$. Therefore, both canonical forms are identical, i.e., $PoS(e) = SoP(e)$.

Situations like this arise when the corresponding *single-effect CCN* is a purely "conjunctive" network (i.e., uses AND gates only).

3.4.2.2 An Example

We now illustrate <u>*Algorithm 3.2*</u> ("*Construction of SoP Canonical Form from PoS Canonical Form*") by applying it to a *PoS* canonical form of the *CCN* in Figure 3.4 on page 75. Specifically, we consider effect e_1 in the *PoS* canonical form and will obtain its *SoP* canonical form, $SoP(e_1)$. Following the steps of <u>*Algorithm 3.2*</u> applied to $PoS(e_1) = c_4.(c_1+c_2+c_6).c_3.c_5$ of <u>four</u> sum-terms $\{c_4, (c_1+c_2+c_6), c_3, c_5\}$. We have:

1. <u>Step 1</u>: We identify <u>four</u> distinct sum-terms; <u>Three</u> of them $\{c_4, c_3, c_5\}$ are of cardinality 1, and <u>one</u> sum-term $\{(c_1+c_2+c_6)\}$ of cardinality 3. Therefore:

a. $PoS_{part1}(e_1) = c_4 \cdot c_3 \cdot c_5$;

b. $PoS_{part2}(e_1) = (c_1+c_2+c_6)$; and,

c. $PoS_{new}(e_1) = c_4 \cdot c_3 \cdot c_5 \cdot (c_1+c_2+c_6)$

Note: $PoS(e_1)$ is equivalent to $PoS_{new}(e_1)$.

2. Step 2: Multiply (left-to-right) $PoS_{part1}(e_1)$ with $SoP_{part2}(e_1)$ to yield:

$$SoP_{temp}(e_1) = c_4 \cdot c_3 \cdot c_5 \cdot c_1 + c_4 \cdot c_3 \cdot c_5 \cdot c_2 + c_4 \cdot c_3 \cdot c_5 \cdot c_6$$

This resulting $SoP(e_1)$ is the sum of three distinct product-terms. Note that We have three product-terms also because $|s_1| \times |s_2| \times |s_3| \times |\overline{s_4}| = 1 \times 1 \times 1 \times 3 = 3$.

3. Step 3: No logical redundancies are detected. Therefore, our *SoP* canonical form remains:

$$SoP_{temp}(e_1) = c_4 \cdot c_3 \cdot c_5 \cdot c_1 + c_4 \cdot c_3 \cdot c_5 \cdot c_2 + c_4 \cdot c_3 \cdot c_5 \cdot c_6$$

4. The resulting $SoP_{temp}(e_1)$ *logical redundancy* free expression, expresses $e_1 = g_1(w_1, w_2, \ldots, w_6)$ by three product-terms. Denote this as $SoP(g_1) = p_1 \cdot p_2 \cdot p_3$ and terminate the process. Here $p_1 = c_4 \cdot c_3 \cdot c_5 \cdot c_1$, $p_2 = c_4 \cdot c_3 \cdot c_5 \cdot c_2$, and $p_3 = c_4 \cdot c_3 \cdot c_5 \cdot c_6$.

The steps of our example are shown in Figure 3.9 on page 95.

Given: (by Algorithm 3.1)

$PoS(e_1) = c_4 \cdot (c_1 + c_2 + c_6) \cdot c_3 \cdot c_5$

Obtain: (using Algorithm 3.2)

$PoS_{new}(e_1) = c_4 \cdot c_3 \cdot c_5 \cdot (c_1 + c_2 + c_6)$

$$SoP(e_1) = c_4 \cdot c_3 \cdot c_5 \cdot c_1 + \quad - p_1$$
$$c_4 \cdot c_3 \cdot c_5 \cdot c_2 + \quad - p_2$$
$$c_4 \cdot c_3 \cdot c_5 \cdot c_6 \quad - p_3$$

(product-terms)

Figure 3.9: An Example Illustrating the Algorithmic Construction of the $SoP(e_1)$ Canonical Form. This form is based on the *CCN* of Figure 3.4 on page 75.

3.5 Algorithmic Generation of Test Requirements Based on Sensitization of Canonical Forms

In Section 3.3, we have seen that a *single-valued CCF* (i.e., f_1: D^n --> D), yielding effect e_1 from *causes* $\{c_1, c_2, \ldots, c_n\}$, can be expressed in terms of *two* canonical forms: $PoS(e_1)$ and its "derivative form" $SoP(e_1)$. These forms provide us with a special kind of "grouping" of causes for e_1 which we have called *sum-terms* and *product-terms*. In this section, we develop a method for generating *test requirements* from these forms based on sensitization techniques from hardware testing technology [Koha 70, FrMe 71].

The basic idea behind *sensitization* of $PoS(e_1)$ and $SoP(e_1)$ is to "exercise" their *sum-terms* and *product-terms one-by-one*. That is, to observe the contribution of *literals* $\{w_1, w_2, \ldots, w_q\}$ of *causes* $\{c_1, c_2, \ldots, c_q\}$ with the *jth sum-terms* (i.e., $s_j = w_1+w_2+, \ldots, +w_q$), of $PoS(e_1) = s_1.s_2., \ldots, .s_u$, with all *causes* $\{c_1, c_2, \ldots, c_n\}$, for *effect* e_1.

In an analogous argument, it is possible to observe the contribution of another set of *literals* $\{w_1, w_2, \ldots, w_r\}$ of *causes* $\{c_1, c_2, \ldots, c_r\}$ via the *kth product-terms* (i.e., $p_k = w_1+w_2+, \ldots, +w_r$), of $SoP(e_1) = p_1.p_2., \ldots, .p_v$, again, with all *causes* $\{c_1, c_2, \ldots, c_n\}$, for *effect* e_1.

It follows from the above that, for an effect $e_1 = g_1(c_1, c_2, \ldots, c_n)$, there is a total number of $(u + v)$ possible ways to "exercise" causes $\{c_1, c_2, \ldots, c_n\}$ viable through the u sum-terms and v product-terms available from the corresponding canonical forms. The following Sections, 3.5.1 and 3.5.2 discuss sensitization of canonical forms in detail.

3.5.1 Stuck-at-0 Sensitization

The main idea behind product-term sensitization applied to a *single-valued CCF*, $e_1 = g_1(c_1, c_2, \ldots, c_n)$, is to detect any *stuck-at-0* faults with respect to effect e_1. The means to do this is to sensitize the *SoP* canonical form in such a way that one *product-term* at a time is tested for its contribution to making the *effect* present.

For example, suppose $p_1 = c_4.c_3.c_5.c_1$ (is the product-term from $SoP(e_1) = p_1+p_2+p_3$, Figure 3.9 on page 95). Moreover,

suppose p_1 is the *product-term* from $\{c_1, c_2, \ldots, c_n\}$ to the effect e_1.

To detect a *stuck-at-0* fault based on p_1 (involving causes $\{c_1, c_3, c_4, c_5\}$), it is necessary to assign a 1 to p_1 (by appropriately assigning 0's and/or 1's to $\{c_1, c_3, c_4, c_5\}$) and 0's to all remaining p_1's (namely p_2 and p_3) in the $SoP(e_1)$.

This *ensures* that all the gates $\{(i_1, 201), (i_2, 202), (i_3, 203), (e_1, 301)\}$ will allow the propagation of $p_1 = \{c_1, c_3, c_4, c_5\}$ to the effect e_1 and that only p_1 will propagate to the effect e_1.

Such an assignment of values $\{p_1, p_2, p_3\}$ causes this *SoP* to be *sensitized* with respect to p_1. - Note: The whole *SoP* is sensitized; not just p_1.

ALGORITHM 3.3: *Stuck-at-0 sensitization of cause product-terms*

Purpose:

Let $G: D^n \dashrightarrow D^m$ be a *m-valued CCF*, where $G = \{g_1, g_2, \ldots, g_m\}$ and each $g_i: D^n \dashrightarrow D$. The set of causes are $\{c_1, c_2, \ldots, c_n\}$ and the set of effects are $\{e_1, e_2, \ldots, e_m\}$ where $e_i = g_i(c_1, c_2, \ldots, c_n)$. Assume, each g_i is in the *SoP* canonical form (*Algorithm 3.2*, Section 3.4.2). That is, $SoP(e_1) = p_1 + p_2 + \ldots + p_v$ where p_j is the *j*th product-term ($p_j = w_1 . w_2 . \ldots . w_r$, Section 3.3.1). Carry out *stuck-at-0-sensitization* for a $PoS(e_1)$ with respect to a p_j.

Procedure:

1. *Case I*: If $v = 1$, we have a *single* product-term $\{p_1\}$ ("special case", Section 3.4.2.1), where $SoP(e_1) = p_1 = w_1 \cdot w_2 \cdot, \ldots, \cdot w_r$ because literals $\{w_1 \cdot w_2 \cdot, \ldots, \cdot w_r\}$ (of causes $\{c_1 \cdot c_2 \cdot, \ldots, \cdot c_r\}$) is *identical* to causes $\{c_1 \cdot c_2 \cdot, \ldots, \cdot c_n\}$. In this *case*, steps (2.) to (3.) are executed *once*.

   Case II: If $v > 1$, we have v product-term $\{p_1, p_2, \ldots, p_v\}$ (Section 3.4.3), then $SoP(e_1) = p_1 + p_2 +, \ldots, + p_v$ where p_j is the jth product-term ($p_j = w_1 \cdot w_2 \cdot, \ldots, \cdot w_r$, Section 3.3.1). Literals $\{w_1 \cdot w_2 \cdot, \ldots, \cdot w_r\}$ of causes $\{c_1 \cdot c_2 \cdot, \ldots, \cdot c_r\}$ is a *subset* of causes $\{c_1 \cdot c_2 \cdot, \ldots, \cdot c_n\}$. In this *case*, steps (2.) to (3.) are executed v times.

2. With respect to the current p_j product-term, assign all causes (with values from $D = \{0, 1, U\}$, Section 3.2.3), in target effect e_1 $\{c_1 \cdot c_2 \cdot, \ldots, \cdot c_n\}$ so that:

   a. Assign the literals of p_j ($p_j = w_1 \cdot w_2 \cdot, \ldots, \cdot w_r$), so that $p_j = 1$. This assignment in turn, is sufficient (due to "null axiom" over "+" Table 3.3 on page 78), to set the entire $SoP(e_1) = 1$.

   b. For every $k \neq j$, assign these literals of p_k product-term (i.e., $p_k = w_1 \cdot w_2 \cdot, \ldots, \cdot w_r$) still left not-assigned with respect to step (2.a.), so that $p_k = 0$.

   c. For every $x \neq i$ ($i = 1, 2, \ldots, m$), evaluate its effect e_x using the $PoS(g_x)$ canonical form (i.e., the xth *single-valued CCF*, g_x: $D^n \dashrightarrow D$ of G: $D^n \dashrightarrow D^m$). Effect e_x may combine *none*, *some*, or *all literals*

$\{w_1, w_2, \ldots, w_n\}$ appearing in the target effect e_i, whose values have been already determined by step (2.a) and step (2.b) above. However, it is possible for $PoS(g_x)$ to also combine literal(s) w_y where $y \neq k$ $(k = 1,2, \ldots, n)$, which is assigned by $w_y = U$. This step is executed $(m-1)$ times and the result of each $PoS(g_x)$ evaluation receives a value form domain $\{0,1,U\}$.

d. Record the above assignments (i.e., $\{0,1\}$), under causes $\{c_1, c_2, \ldots, c_n\}$ along with the value of the resulting target effect $e_i = 1$, as well as $(m-1)$ remaining values due to step (2.c.) from $\{0,1,U\}$, for a later reference in the process.

3. Repeat step (2.), until all p_j product-terms (i.e., $j = 1,2, \ldots, v$) of the $SoP(e_1) = p_1 + p_2 +, \ldots, + p_v$ have been considered. Clearly, form this step (3.) we would go to step (2.) $(v-1)$ times.

4. Observe that a total number of v sets of assignments have been recorded and the procedure is terminated.

Algorithm 3.3, was carried out for *stuck-at-0* sensitization for the *ith single-valued CCF* (i.e., $g_i: D^n \dashrightarrow D$), cause product-terms. However, if we had dealt with *m valued CCF's* (i.e., $G: D^n \dashrightarrow D^m$) then from step (3.) we would go to step (1.) $(m-1)$. Every time the next *xth single-valued CCF*, $e_x = g_x(c_1, c_2, \ldots, c_n$, where $x \neq i$ $(i = 1,2, \ldots, m)$, is considered.

3.5.1.1 An Example

We illustrate *Algorithm 3.3* ("*Stuck-at-0 Sensitization of Cause Product-Terms*", Section 3.5.1), being applied to a *single product-term* of a $SoP(g_1)$ canonical form (additional product-terms are treated similarly). Specifically, we consider the $SoP(g_1) = p_1+p_2+p_3$ of Section 3.4.2.2, shown in Figure 3.9 on page 95. Applying *Algorithm 3.3* to *one* such product-term, $p_1 = c_4.c_3.c_5.c_1$, we have:

1. Step 1: We have *Case II*. Therefore, exercise $p_1 = c_4.c_3.c_5.c_1$. This is accomplished by executing [Steps 2. to 3.] once. Note that the participating literals $\{w_1,w_3,w_4,w_5\}$ of causes $\{c_1,c_3,c_4,c_5\}$ is a *subset* of $\{w_1,w_2, \ldots ,w_6\}$.
2. Step 2: Focusing on p_1 (that incorporates only $\{w_1,w_3,w_4,w_5\}$), we assign all literals in the target effect e_1, $\{w_1,w_2, \ldots ,w_6\}$, with values from $\{0,1,U\}$, so that:
   a. It is required that $p_1 = 1$, which is achieved by setting $w_1 = 1$, $w_3 = 1$, $w_4 = 1$, and $w_5 = 1$ (i.e., $p_1 = c_4.c_3.c_5.c_1$). Therefore, $SoP(g_1) = 1$, since $SoP(g_1) = 1+p_2+p_3$ and so $e_1 = 1$.
   b. It is required that $p_2 = 0$, and $p_3 = 0$. From Figure 3.9 on page 95, we note that $p_2 = c_4.c_3.c_5.c_2$, and $p_3 = c_4.c_3.c_5.c_6$. To satisfy this requirement, we only have to set $c_2 = 0$, and $c_6 = 0$. Note that, for the remaining literals of p_2 and p_3 (i.e. $\in \{w_3,w_4,w_5\}$), their values have already been determined by step (2.a.).

c. We have to evaluate the $PoS(g_2)$ based on the assignments of its literals (if any). However, $PoS(g_2) = c_4.(c_1+c_2+c_6).(c_3'+c_5')$, as shown in Figure 3.8 on page 91. Note that its values have been already determined by steps (2.a.) and (2.b.) above. Thus, $PoS(g_2) = 1.(1+0+0).(1'+1')$ and we can easily verify that $e_2 = 0$.

d. We now record all the values found above for a later reference.

3. Step 3: Not applicable, since we sensitized only *one* product-term, $\{p_1\}$.

4. Step 4: We observe that *one* set of values has been recorded, shown in Figure 3.10 and terminate the process.

Causes						*Effects*	
c_1	c_2	c_3	c_4	c_5	c_6	e_1	e_2
1 *	0	1 *	1 *	1 *	0	1 *	0

Figure 3.10: An Example Illustrating the Stuck-at-0 Sensitization of a Cause product-term. For this illustration we have used the product-term p_1 of $SoP(g_1)$ shown in Figure 3.9 on page 95. An asterisk ("*") is used to "flag" the *forcing-conditions (causes)* and the *target effect*.

3.5.2 Stuck-at-1 Sensitization

Similarly, as in Section 3.5.1, the main idea behind sum-term sensitization, applied to a *single-valued CCF*, i.e., $e_1 = g_1(c_1, c_2, \ldots, c_n)$, is to detect any *stuck-at-1* faults with respect to effect e_1. We do this by sensitizing the *PoS* canonical form in such a way that one *sum-term* at a time is tested for its contribution to making the *effect* not-present.

For example, suppose $s_3 = (w_3+w_5)$ (is the sum-term $PoS(e_2) = s_1 \cdot s_2 \cdot s_3$, Figure 3.8 on page 91) of the *standard CCN* of Figure 3.4 on page 75), and is the *only one* sum-term from $\{c_1, c_2, \ldots, c_n\}$ to the effect e_2.

To detect a *stuck-at-1* fault based on s_3 (involving literals $\{w_3, w_5\}$), it is necessary to assign a 0 to s_3 (by appropriately assigning 0's and/or 1's to $\{w_3, w_5\}$). And 1's to all reraining s_j's (namely s_1 and s_2) in the $PoS(e_2)$.

This *ensures* that all the gates involved $\{(i_1,201), (i_2,202), (i_3,203), (i_4,204), (e_1,301)\}$, will allow the propagation of $\{w_3, w_5\}$ to the effect e_2 and that only $\{w_3, w_5\}$ will reach the effect e_2. The assignment of values $\{0,1\}$ to $\{w_3, w_5\}$ of s_3, makes this sum-term to be *sensitized*.

ALGORITHM 3.4: *Stuck-at-1 sensitization of cause sum-terms*

Purpose:

Similar to *Algorithm 3.3*, let $G: D^n \dashrightarrow D^m$ be a *m-valued CCF*, where $G = \{g_1, g_2, \ldots, g_m\}$ and each $g_1: D^n \dashrightarrow D$. The set of causes are $\{c_1, c_2, \ldots, c_n\}$ and the set of effects are $\{e_1, e_2,$

... ,e_m} where $e_1 = g_1(c_1, c_2, \ldots, c_n)$. Assume that each g_i is in the *PoS* canonical form (*Algorithm 3.1*, Section 3.4.1). That is $PoS(e_1) = s_1.s_2., \ldots, .s_u$ where s_j is the *j*th sum-term ($s_j = w_1{+}w_2{+}, \ldots, {+}w_q$), Section 3.3.1. Carry out *stuck-at-1 sensitization* for a $SoP(e_1)$ with respect to a s_j.

Procedure:

1. With respect to the current s_j sum-term assign all causes (with values from $D = \{0,1,U\}$, Section 3.2.3), in target effect e_1, $\{c_1, c_2, \ldots, c_n\}$, so that:
   a. Assign the literals of s_j ($s_j = w_1{+}w_2{+}, \ldots, {+}w_q$), so that $s_j = 0$. This assignment, in turn, is sufficient (due to "null axiom" over "." Table 3.3 on page 78), to set the entire $PoS(e_1) = 0$.
   b. For every $k \neq j$, assign these literals of s_k sum-term (i.e., $s_k = w_1{+}w_2{+}, \ldots, {+}w_q$), still left not-assigned with respect to step (1.a.), so that $s_k = 1$.
   c. This step is *identical* to [Step-2-c.] of *Algorithm 3.3* in Section 3.5.1.
   d. Record the above assignments (i.e., $\{0,1\}$), under causes $\{c_1, c_2, \ldots, c_n\}$ along with the value of the resulting target effect $e_1 = 0$, as well as $(m-1)$ remaining values due to step (1.c.) from $\{0,1,U\}$, for a later reference.
2. Repeat step (1.), until all s_j sum-terms ($j = 1,2, \ldots, u$) of $PoS(e_1) = s_1.s_2., \ldots, .s_u$ have been considered. Clearly, from this step (2.) we would go to step (1.) $(u-1)$ times.

3. Observe that a total number of u sets of assignments have been recorded and then the process is terminated.

Again, *Algorithm 3.4*, was carried out for *stuck-at-1* sensitization of the *ith single-valued CCF* (i.e., g_i: D^n --> D), cause sum-terms. However, if we had dealt with *m-valued CCF's* (i.e., G: D^n --> D^m) then from step (2.) we would go to step (1.) *(m-1)*. Every time the next *xth single-valued CCF*, $e_x = g_x(c_1, c_2, \ldots, c_n$, where $x \neq i$ $(i = 1,2, \ldots, m)$, is considered.

3.5.2.1 An Example

We now illustrate *Algorithm 3.4* ("*Stuck-at-1 Sensitization of Cause Sum-terms*") applied to a *single sum-term* of a $PoS(g_1)$ canonical form (remaining sum-terms are treated similarly). For this illustration, we consider the $PoS(g_2) = s_1.s_2.s_3$ of Section 3.4.1.1, shown in Figure 3.8 on page 91. Applying *Algorithm 3.4* to *one* such sum-term, $s_3 = (c_3'+c_5')$, we have:

1. Step 1: We exercise $s_3 = (c_3'+c_5')$. Note that the participating literals $\{w_3, w_5\}$ of causes $\{c_3, c_5\}$ is a *subset* of $\{w_1, w_2, \ldots, w_6\}$. Thus, focusing on s_3 (that incorporates only $\{w_3, w_5\}$), assign all literals in the target effect e_2, $\{w_1, w_2, \ldots, w_6\}$, with values from $\{0,1,U\}$, so that:
   a. It is required that $s_3 = 0$, which is achieved by setting $w_3 = 1$, and $w_5 = 1$, since $s_3 = (c_3'+c_5')$. Therefore, $PoS(g_2) = 0$, since $PoS(g_2) = s_1.s_2.0$ and so $e_2 = 0$.

b. Both $s_1 - 1$, and $s_2 - 1$ must each be set to 1. From Figure 3.8 on page 91, we confirm that $s_1 - c_4$, and $s_2 - (c_1+c_2+c_6)$. To satisfy this requirement, we need to set $c_4 - 1$, for s_1. As far as the s_2 (since we treat the OR gate as being an "Exclusive-OR", see Section 6.2) it requires that at most one of $\{c_1, c_2, c_6\}$ be set to 1. Thus, we choose to set the first: $c_1 - 1$ and the remaining: $c_2 - 0$, $c_6 - 0$.

c. We have to evaluate the $PoS(g_1)$ based on the assignments of its literals (if any) of steps (1.a.) and (1.b.). However, $PoS(g_1) - c_4.(c_1+c_2+c_6).c_3.c_5$, shown in Figure 3.5 on page 76. Thus, $PoS(g_1) - 1.(1+0+0).1.1$, therefore, we can easily verify that $e_1 - 1$.

d. We now record all the values found above for a later reference.

2. Step 2: Not applicable, since we have sensitized only *one* sum-term, $\{s_3\}$.

3. Step 3: We observe that one set of values has been recorded, shown in Figure 3.11 on page 106 and the process terminated.

Causes						*Effects*	
c_1	c_2	c_3	c_4	c_5	c_6	e_1	e_2
1	0	1 *	1	1 *	0	1	0 *

Figure 3.11: An Example Illustrating the Stuck-at-1 Sensitization of a Cause sum-term. For this illustration we have used the sum-term s_3 of $PoS(g_2)$ shown in Figure 3.8 on page 91. Again, an asterisk ("*") is used to "flag" the *forcing-conditions (causes)* and the *target effect*.

3.6 Incremental Generation of Test Requirements by Removal of Test Redundancy

Section 3.3 and Section 3.4 were carried out with respect to a *single-valued CCF*. For each effect e such that $|PoS(e)| = u$ and $|SoP(e)| = v$, there are $u + v$ tests obtained by sensitizing causes. Note that u and v represent the number of *sum-terms* and the number of *product-terms* of a *single-valued CCF*. Note that $PoS(e) = s_1 \cdot s_2 \cdot \ldots \cdot s_u$ of u sum-terms and $SoP(e) = p_1 + p_2 + \ldots + p_v$ of v product-terms.

Algorithm 3.1 and *Algorithm 3.3* both deal with a type of information redundancy called *logical redundancy* (see Section 3.4). However, the $u + v$ test requirements generated by *Algorithms 3.3* and *3.4* may contain instances of identical (thus, redundant) *cause/effect* assignments. We call this type of redundancy *test requirement redundancy*.

3.6.1 Test Requirement Redundancy: Checking and Removal

We now present a strategy for detecting and avoiding *redundant test requirements*. Our algorithm (described next), is general enough to detect redundant test requirements manually (one particular *single-valued CCF* is chosen to be sensitized more than-once) as well as to support situations where the fundamental assumption of *CCN* integrity is violated (see Section 3.4).

Here, *test requirements* are generated one per sensitization of each *single-valued CCF*. However, before a test requirement is added to the suite of test requirements, the test requirement is compared to all previously generated test requirements. If the newest test requirement is identical to an already existing one it is *ignored*, otherwise it is *added* to the test suite.

ALGORITHM 3.5: *Redundancy checking and removal*

Purpose:

Given G: $D^n \dashrightarrow D^m$ the set of *m-valued CCF's* where $G = \{g_1, g_2, \ldots, g_m\}$ and each g_i: $D^n \dashrightarrow D$. The set of *causes* are $\{c_1, c_2, \ldots, c_n\}$ and the set of *effects* are $\{e_1, e_2, \ldots, e_m\}$ where $e_i = g_i(c_1, c_2, \ldots, c_n)$. Obtain a test suite R of distinct test requirements from sensitizing the canonical forms of G.

Procedure:

1. Apply *stuck-at-0* sensitization to generate u test requirements $\{r_1, r_2, \ldots, r_u\}$ from the $SoP(g_i)$ using *Algorithm 3.3*.

2. *Open* the test suite R and *insert* all u test requirements.

   Note: No *test requirement redundancy* can be generated by step (1.) because *logical redundancy* removal has been applied earlier (*Algorithm 3.3* - with respect to a *single-valued CCF*, g_1: $D^n \dashrightarrow D$).

3. For each g_j, (j = 1,2, ... ,m; *for* $j \neq 1$) do:

   a. Apply *stuck-at-0* sensitization to generate u test requirements $\{r_1, r_2, \ldots, r_u\}$ from the $SoP(g_j)$ using *Algorithm 3.3*.

   b. For each *new* test requirement r_k, check whether r_k appears in R already. If not, $R \leftarrow R \cup \{r_k\}$.

4. For each g_j, (j = 1,2, ... ,m) do:

   a. Apply *stuck-at-1* sensitization to generate v test requirements $\{r_1, r_2, \ldots, r_v\}$ from the $PoS(g_j)$ using *Algorithm 3.4*.

   b. For each *new* test requirement r_k, check whether r_k appears in R already. If not, $R \leftarrow R \cup \{r_k\}$.

3.6.1.1 An Example

We now illustrate the above *Algorithm 3.5* applied to two *single-valued CCF's*, (G: $D^6 \dashrightarrow D^2$) realizable by the *standard CCN* shown in Figure 3.4 on page 75.

For our example, we consider the *test requirements* derived from $SoP(g_1) = p_1 + p_2 + p_3$ shown in Figure 3.9 on page 95, and those derived from $PoS(g_2) = s_1 . s_2 . s_3$ shown in Figure 3.8 on page 91. Applying *Algorithm 3.5* to $SoP(g_1)$ and $PoS(g_2)$ yields:

1. Step 1: Generate *stuck-at-0* test requirements to an *SoP* canonical form. Here, $SoP(g_1)$.

2. Step 2: Open the test suite R (R <- *null*); and insert all three test requirements: $R \leftarrow R \cup \{p_1, p_2, p_3\}$. These are:

Test Suite R:

Causes						*Effects*		*Sensitized Product-terms*
c_1	c_2	c_3	c_4	c_5	c_6	e_1	e_2	
1*	0	1*	1*	1*	0	1*	0	- p_1
0	1*	1*	1*	1*	0	1*	0	- p_2
0	0	1*	1*	1*	1*	1*	0	- p_3

3. Step 3: Here, $i = 1$.

4. Step 4: Apply *Algorithm 3.4* to $PoS(g_2)$. The three *new* generated test requirements are kept in a *buffer* as follows:

Buffer:

Causes						*Effects*		*Sensitized Sum-terms*
c_1	c_2	c_3	c_4	c_5	c_6	e_1	e_2	
1	0	0	0*	1	0	0	0*	- s_1
0*	0*	0	1	1	0*	0	0*	- s_2
1	0	1*	1	1*	0	1	0*	- s_3

Redundancy checking: The first two test requirements on the *buffer* (e.g., $\{s_1, s_2\}$) are *new* and are added to the test suite: $R \leftarrow R \cup \{s_1, s_2\}$. The third test requirement on *buffer* (e.g., s_3) is identical to the test requirement

already in the test suite (see p_1 in Step 2). Therefore s_3 is ignored and is not added in the test suite R.

The procedure terminates, resulting in the *final* test suite shown in Figure 3.12.

Test Suite R:

Causes						*Effects*		*Sensitized Product/Sum Terms*
c_1	c_2	c_3	c_4	c_5	c_6	e_1	e_2	
1 *	0	1 *	1 *	1 *	0	1 *	0	$- p_1$
0	1 *	1 *	1 *	1 *	0	1 *	0	$- p_2$
0	0	1 *	1 *	1 *	1 *	1 *	0	$- p_3$
1	0	0	0 *	1	0	0	0 *	$- s_1$
0 *	0 *	0	1	1	0 *	0	0 *	$- s_2$

Figure 3.12: An Example Illustrating the Test Requirement Redundancy Checking and Removal. This example uses $\{SoP(g_1), PoS(g_2)\}$ canonical forms from the *CCN* shown in Figure 3.4 on page 75.

3.7 Summary: Incremental Sensitized Test Design

In this Chapter, we have introduced *standard CCN's* and *standard CCF's* involving a *ternary* logical domain $D = \{0,1,U\}$ for representing a functional specification as a set of logical relationships between "*causes*" and "*effects*". A *CCN* is the

starting point for our incremental methodology for generating test requirements.

Concepts from hardware testing such as fault detection by path sensitization have inspired our approach to test design. In it, we sensitize functional specifications represented in canonical forms of *CCF's*.

We also presented algorithms for the construction of canonical forms, generation of test requirements based on sensitization of canonical forms, and incremental generation of test requirements by redundancy removal. As an illustration, these algorithms were applied to a subset of Naur's TRP. More details and an application of the method to the complete problem appear in Appendix A.

In the next Chapter, we describe a prototype system which implements our approach. This tool is described from the system's and user's points of view.

Chapter IV
A PROTOTYPE TOOL FOR INCREMENTAL TEST DESIGN

In this Chapter, we describe in detail our INcremental SEnsitized Test Requirement Design (abbreviated DINSERT) prototype system. DINSERT is a user-friendly menu-driven interactive prototype system that implements our incremental sensitized test design methodology described in Chapter III.

Here we discuss the design rationale for DINSERT, give a design overview followed by detailed design considerations and assessment of the DINSERT prototype. In Appendix A, we illustrate the features of this prototype by applying DINSERT to Text Reformatter Problem (abbreviated TRP).

4.1 Design Rationale for DINSERT

As black-box methodologies have gained a strong appreciation during the last decade, the need for black-box automated tools has become greater and greater. Such an ideal tool should be able to uniquely represent specifications, and be attractive and easy to use. It should have systematic and clear steps.

DINSERT is a user-friendly menu-driven interactive prototype system. It is a direct implementation of our incremental sensitized test design methodology, described in Chapter III. The

algorithms are implemented in DINSERT exactly as described in Chapter III. But in DINSERT, we use a more suitable notation for the computer implementation both to represent the *CCN* and to process the *CCF canonical forms* (see Appendix A, Sections A.2 and A.4).

The target user of our tool is the software engineer (developer and/or tester). The developer would use our tool in early stages of software development to detect ambiguities, inconsistencies and omissions in the specifications - well before any target software is produced. In this case our tool would serve as an advisor. But the tool's primary user was meant to be the software tester - provided with useful guidance to the systematic test requirement generation.

In practice, when a large scale project (e.g., testing couple of hundreds of *effects* capable of generating thousands of *test requirements*) is conducted, one may argue how practical (efficient) the interactive environment of DINSERT is. In our approach, practicality is not measured by the number of test requirements being generated. For example, 100 test requirements could be the overall contribution of several effects or the contribution of a single effect only. Therefore, depends on the size and the complexity of the *CCN* and how is treated.

In general, however, the above argument is true. But in our incremental approach we believe that the most effective environment to test requirement generation, for our experimental prototype tool, would be the interactive environment instead of

using batch. The *CCF canonical forms* (Appendix A, Sections A.4.1 and A.4.2), menu command responses (see Appendix B) and various other explanations during the interaction ("test-suite-window", see Sections A.4.1.1 and A.4.2.1) communicate meaningful information to the tester. As a result the tester may assess well in advance (or as soon as become evident) any anomalies in the run. Accordingly, he may continue the run. Otherwise, he may immediately terminate the run to avoid unnecessary expense. So gains in efficiency.

In principle, a large problem can always be divided into smaller problems ("divide-and-conquer"). But in DINSERT this works only when you deal with problems with obvious partitions in their *CCN*. That is to say the *CCN* is divided into independent subnetworks (e.g., TRP, see Figure A.2 on page 180). Then, each such subnetwork may be treated independently (i.e., by a separate DINSERT run) and the combined test requirements may equal to those obtained by the *CCN* (i.e., treated as a whole). Otherwise, the *test requirement redundancy* removal (see Section 3.6) is not guaranteed because no records of the test requirements of the other subnetworks exist since each subnetwork is associated with its own test suite.

Currently, DINSERT is running in CMS on the University's research mainframe. Its portability, however, to another computer system is discussed in Section 4.3.3.

4.2 Design Overview

In this section, we briefly describe how the DINSERT subsystems are linked to each other and how the processing results are recorded. Figure 4.1 on page 116, clearly demonstrates that the processing of each subsystem depends on the data made available to it by its predecessor. Thus, an interface (subsystem-handshake) is established among the five subsystems of DINSERT.

4.2.1 General Subsystem Organization

DINSERT has five subsystems: (1) the *CCN editor*; (2) the *CCF generator*; (3) the *CCF sensitizer*; (4) the *test requirement generator*; and (5) the test suite incrementor. The overall subsystem organization is shown in Figure 4.1 on page 116. Brief functional descriptions follow:

1. *CCN editor*: Allows the user to edit the *CCN* (e.g., Figure A.2 on page 180). Basically, *CCN editor* carries out three tasks: (1) it allocates storage for the *CCN*; (2) it assigns gate types {AND,OR,NOT}; and (3) it connects gates together. The output from this subsystem is the *GTF* representation of the *CCN* (e.g., Figure 4.3 on page 129). This subsystem is implemented by the CCNEDIT module.
2. *CCF generator*: Generates the canonical forms of all the *single-valued CCF's* of a *CCN* (e.g., the *PoS* and *SoP* canonical forms - see Section A.6.1). This subsystem is implemented by the CCFGEN module.
3. *CCF sensitizer*: Generates the *stuck-at-0* and *stuck-at-1* *compact test libraries*. These are compact (matrix)

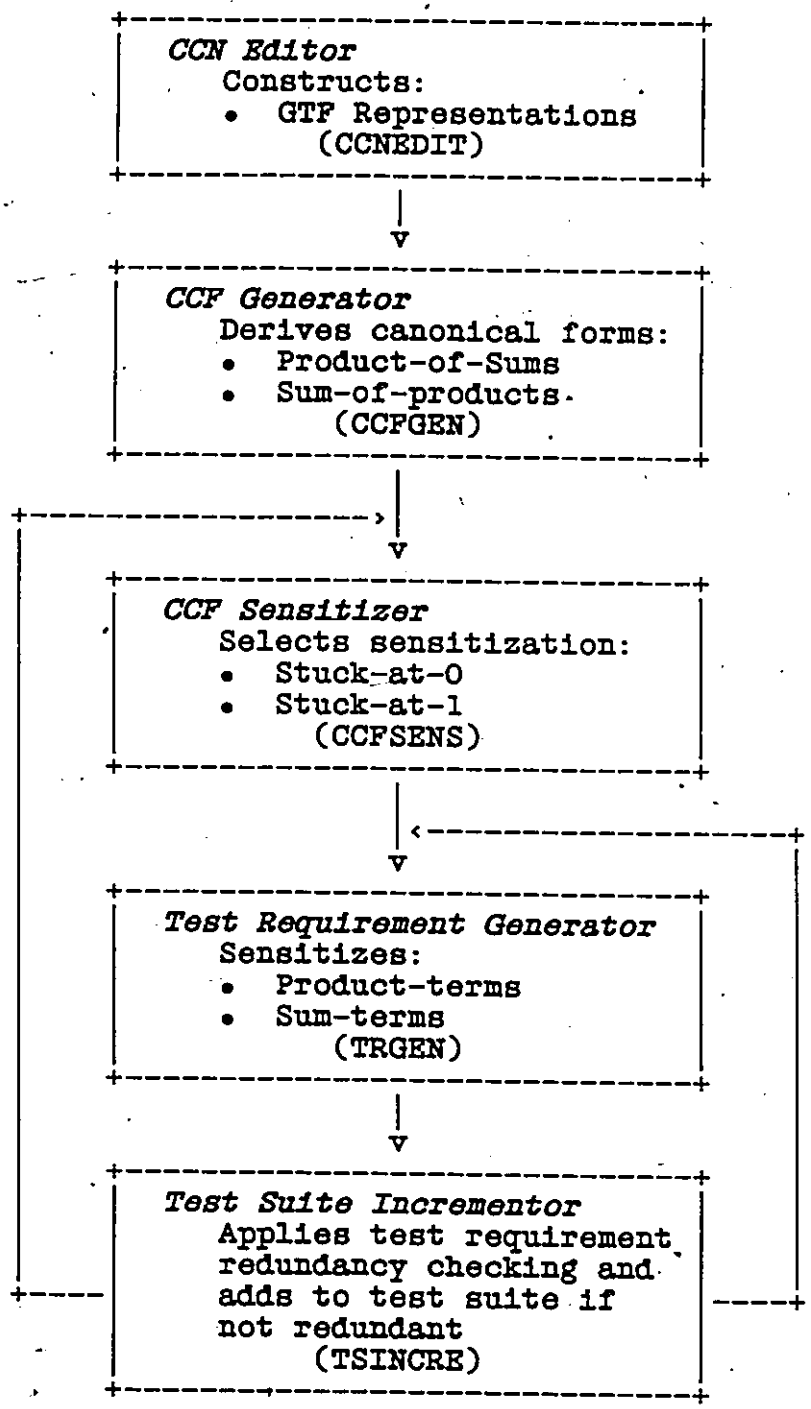


Figure 4.1: Simplified High-Level Flow-Graph of DINSERT Subsystem Organization.

representations of *CCF's* in terms of *ternary* values *{0,1,U}* (e.g., see Section 4.3.2.3, Table 4.5 on page 130 and Table 4.6 on page 131). This subsystem is implemented by the CCFSENS module.

4. *Test requirement generator*: Is the subsystem that generates test requirements. Based on the type of sensitization (i.e., *stuck-at-0* or *stuck-at-1*) specified by the user, the proper *product-term(s)* or *sum-term(s)* are selected. Then, using the proper *compact test library*, the corresponding test requirements are constructed. And they are kept in a *buffer* (i.e., a *temporary test suite*). This subsystem is implemented by the TRGEN module.

5. *Test Suite Incrementor*: Updates the test requirement data base. Redundancy checking is applied to the outputs of the *test requirement generator*, Step (4.). Non-redundant test requirements are transferred into the test suite; otherwise, they are ignored. This subsystem is implemented by TSINCRE module.

4.2.2 Major Files and Data Structures

The file organization of DINSERT is shown in Figure 4.2 on page 118. It has (1) *direct access input files*: called FILE CCNGDEF and FILE WORKPAD (defined in Table 4.1 on page 119); and (2) *sequential access output files*: called LOGS LISTING and TSUITE LISTING (discussed in Section 4.3.2).

The subsystem interface data are kept in the memory (i.e., using one, two, or three dimensional arrays). Every subsystem

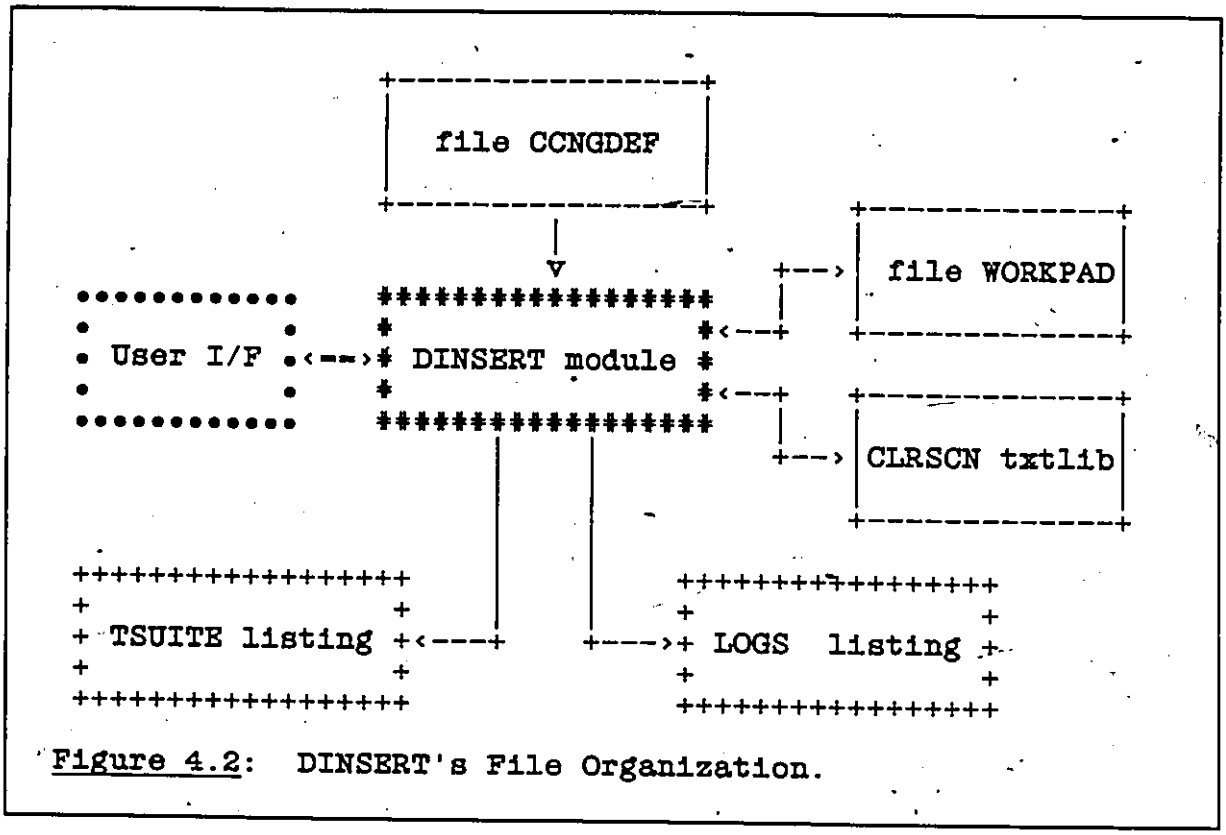


Figure 4.2: DINSERT's File Organization.

accepts data from the previous subsystem and generates data that in turn become the input data of the next subsystem. We classify these data into two categories:

1. First, data that have to do with the subsystem interface, as mentioned above, are meaningful to us because they provide a kind of log and trace capability of the data flow per subsystem. We store that information in a file called LOGS LISTING.
2. The second category has the actual output data organized into a test suite. We store the test suite in a file called TSUITE LISTING.

Data kept in LOGS LISTING and in TSUITE LISTING are illustrated in Section 4.3.2 and in Appendix A (Sections A.6.1 and A.6). Again, these illustrations are based on the TRP.

Table 4.1: DINSERT Input File Description.

Listings	Descriptions
FILE CCNGDEF : 1 . AND 2 + OR 3 ' NOT 4 • BUF	Is a *direct access read-only* file that contains the gate definitions used in CCN's. Has 4 records of logical record-length 10.
FILE WORKPAD : xxxxxxxxxx	A *direct access read/write* file of 1 record with logical record length 10 used as *work-area* by DINSERT.
CLRSCN ASSEMBLE : PRINT NOGEN DEBUT CLRSCN LA R1,CMD SVC 202 DC AL4(*+4) SR R15,R15 SCHLUSS CMD DS 0D DC CL8'EXECSERV' DC CL8'CLEARSCN' DC 8X'FF' END CLRSCN	Is a system dependent user-defined MACRO. When this macro is executed clears the screen. For our AMDAHL 470/V8, we wrote this macro in IBM 360/370 ASSEMBLY. Then convert it into a system TXTLIB (CLRSCN) enabling interface to 370/OS. It is called via DINSERT by: SUBROUTINE CLEAR CALL CLRSCN RETURN END

4.2.2.1 CCN Internal Representation

The user specified data of the *CCN Editor* (CCNEDIT module) are stored in memory using arrays. These arrays are organized to make a "base" for the *CCN* data. These data are divided (distributed)

Table 4.2: "CCN Connection Summaries" of the CCN Internal Representation.

The data in this table are based on the TRP whose *CCN* is shown in Figure A.2 on page 180. GateTypes (1,2,3,4) are defined in FILE CCNGDEF shown in Table 4.1 on page 119.

GateID	GateType	FanIn	FanOut	Status
101:	4	0	2	0
102:	4	0	2	0
103:	4	0	1	0
104:	4	0	1	0
105:	4	0	1	0
106:	4	0	2	0
107:	4	0	1	0
108:	4	0	1	0
. . . (91) CAUSE Records SUPPRESSED . . .				
201:	2	3	2	0
202:	1	2	2	0
203:	1	2	2	0
204:	3	1	1	0
205:	3	1	1	0
206:	3	1	1	0
. . . (93) INTERM Records SUPPRESSED . . .				
301:	1	2	0	0
302:	1	2	0	0
303:	1	2	0	0
304:	1	2	0	0
305:	1	2	0	0
. . . (94) EFFECT Records SUPPRESSED . . .				

into two parts: "part one" and "part two" (similar to data base "relations"). "Part one" relates data about the "CCN-connection summaries" (Table 4.2 on page 120). "Part-two" relates data about the "CCN gate FanIN/FanOut" lists (Table 4.3 on page 122). Communication between these two "parts" - and/or any cross reference - is carried out using a "common-key" the "GateID". So to obtain the *GTF representation* of a *CCN* (see Section 3.4.1) is a simple task. It deals with references between these two "parts", while interpretation of the various data entries is based on data stored in a file called FILE CCNGDEF (Table 4.1 on page 119). The following example illustrates these points:

EXAMPLE: Suppose we want to obtain the *GTF* of gate 201 of the *CCN* of Figure A.2 on page 180. We proceed as follows:

1. Go to entry 201 ("GateID" = 201, see Table 4.2 on page 120) and get the (GateType, FanIn, Status). Check the status: If *status = 0*, then this gate is connected properly in the *CCN*. If *status ≠ 0* (e.g., 1 or 2), then *warning* messages No.31 or No.32 are issued followed by the *prompt* No.1 (see Appendix B). Then *prompt* to specify a connection (e.g., see Appendix A, Section A.3.4).
2. Go to Table 4.3 on page 122, and read the "FanIn-List" corresponding to "GateID" = 201, which is the string: 103104108.
3. Use the "GateType", which is 2, and the direct access the file FILE CCNGDEF, shown in Table 4.1 on page 119), to find out that gate 201 is an OR gate. It is represented by the ("+") operator.

4. Use this operator. Distributed over the 103104108 delimited by parentheses to produce the *GTF* of gate 201. It finally becomes: 201 = (103 + 104 + 108).

Similarly, the remaining *GTF representations* are shown in Figure 4.3 on page 129. Note that, FILE CCNGDEF is an independent input file used by various DINSERT subsystems.

Table 4.3: "CCN Gate FanIn/FanOut" of the CCN Internal Representation.

The data in this table are based on the TPR whose *CCN* is shown in Figure A.2 on page 180.

	FanIn-List	*FanOut-List*
101:		301302
102:		204302
103:	*Work-area to store*	201
104:	*cause constraints.*	201
105:	*For now, this area*	203
106:	*is empty (Blanked).*	202205
107:		203
108:		201

. . . (91) CAUSE Records SUPPRESSED . . .

201:	103104108	305202
202:	106201	303304
203:	105107	303206
204:	102	301
205:	106	305
206:	203	304

. . . (93) INTERM Records SUPPRESSED . . .

301:	101204	*Work-area to store*
302:	101102	*effect constraints.*
303:	202203	*For now, this area*
304:	202206	*is empty (Blanked).*
305:	201205	

. . . (94) EFFECT Records SUPPRESSED . . .

While *GTF's* provide the data interface between CCNEDIT and CCFGEN, a copy of all three data structures ("part one", "part two", and *GTF representations*) is registered in the logs (LOGS LISTING). Table 4.2 on page 120 and Table 4.3 on page 122 illustrate the two relations of the CCN data. Again, these data are based on the TRP.

4.3 Detailed Design Considerations

In this section, we discuss some of the DINSERT design considerations. We focus on the design features of the user interface, logging and the system interface.

4.3.1 User Interface Design Features

DINSERT offers a user-friendly menu-driven interface that help the user to develop test requirements in an interactive environment. This design has two parts. The *top part* and the *bottom part*:

1. *Top part*: A *screen-panel* starting at the top of the screen that contains two kinds of information:
   a. *Static*: Global information about DINSERT; minimum and maximum range values, global summary statistics, etc. Also identification of the current subsystem execution.
   b. *Dynamic*: Displays the valid ranges of data, local summary statistics and, in general, any field in the panel containing data.

2. ***Bottom part***: A ***menu*** immediately following the *top part* is displayed in a horizontal line. The user selects one of the pre-defined entries. Menu *prompts* are related to data entry, explanations, help, screen management, subsystem control flow, etc.

An example of this type of ***screen-panel/menu*** user interface design is shown in Appendix A (e.g., browse through Sections A.3, A.4, etc.). Again, the data shown in these sections correspond to TRP.

4.3.1.1 DINSERT Menu Command Responses

A list of DINSERT menu command responses are shown in Table 4.4 on page 125. The same Table also enumerates the responses (referenced by a number from No.1 to No.64) that could be generated when a command is executed through the environment of the corresponding subsystem.

Furthermore, this Table lists the menu command responses for each DINSERT subsystem (Figure 4.1 on page 116). The texts to menu command responses are given in Appendix B. We divide the responses into two categories: (1) *Prompts* to the user from the DINSERT subsystems to enter data; and (2) those sent from DINSERT to the user that are classified as *warning*, *axiom explanation message*, *error* and *gate connection statements*. The DINSERT menu command responses follow a simple syntax. This syntax is composed of two parts: the ***response head*** immediately followed by the ***response text***. A legend for the various *response heads* appears at the bottom of Table 4.4 on page 124.

Table 4.4: Responses to Menu Commands Organized by DINSERT Subsystem.

There are *five* kinds of responses: *Prompts*, *warnings*, *messages*, *gate connectors*, and *Errors*. Note: this table should be read in conjunction with Appendix B.

Subsystem:	*Responses with respect to the subsystem*:
CCNEDIT --	1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13, 16,17,18,19,20,26,27,28,29,30,31,32,33, 34,35,36,37,38,39,40,41,42,43,44,45,46, 47,48,49,51
CCFGEN --	1,16,17,18,52,53,54,55,56,57,58,
CCFSENS --	1,14,16,17,18,19,20,21,26,27,28,29,30, 59,60,61,62,63,64
TCGEN --	1,15,16,17,18,22,23,24,25
TSINCRE --	1,16,17,18

Legend to *response heads*:

Prompt	Warning	Axiom MSG	Error	Gate Conn
---›	***›	**MSG**	+++›	---›

In Appendix B, a generalized list of the DINSERT responses is given. There (nnn) stands for the three digit gate identification number and (AAA) for the gate type {AND,OR,NOT} (defined in Table 3.1 on page 67). Also (nn) stands for a two-digit number used as a counter etc. In some responses, however, actual data were entered on purpose to emphasize the semantics as well as to help to explain the origin of the response (e.g., No.34, No.34, No.35

etc.). On the other hand, constant values denote the current capability of the prototype (e.g., FanIn and FanOut are pre-set to a bound of 10 to {AND,OR} gates, maximum size of the *CCN* areas are set to 99, etc.)

4.3.1.2 Reporting and Display Features

The user interface is further enhanced with a *user defined macro*. The execution of this *macro* causes the screen of your terminal to be cleared between interactions. We made this *macro* to be a TXTLIB macro. It is called CLRSCN TXTLIB. called CLRSCN TXTLIB. The source (IBM 360/370 ASSEMBLY language) that this CLRSCN TXTLIB has been created from is shown in Table 4.1 on page 119. In this way DINSERT can interface with the 360/370 *Operating System*. Both input and output files are defined in the OPEN subroutine of the CCNEDIT module (e.g., Subroutine OPEN: CCNEDIT / B$UTIL / OPEN, Table 4.7 on page 132).

Document the Test Suite:

All test requirements are placed in a test suite in the order generated. The order is not essential and is user specified. The test suite resembles a high level organization to allow the user at a glance to locate a test requirement with detailed information of its origin. The major features of our test suite are:

1. *Test suite header*: (1) Contains a 25 character field for the test suite *identification*; (2) three fields to summarize the number of *causes*, *intermediate-causes* and

effects involved in the *CCN*; and (3) *causes/effects* headings to aid cross references between test requirements values (e.g., *{0,1,U}*) assigned to corresponding *causes* and *effects*.

2. *Test suite body*: Is the area where the test requirements are incrementally stored. To improve in legibility, the test requirements of deferent *effects* are separated by a "doubled" line (e.g., "======"). In turn, the test requirements due to the same *effect* are separated by a "single" line (e.g., "------").

Every test requirements occupies two lines of the test suite space. Each of these lines states different information of the test requirement being reported:

a. The *first* line is divided into four areas: (1) the *global test requirements counter*; (2) the *local test requirement counter*, followed by the *constant* number of test requirements of this effect; (3) *cause test requirements string*; and (4) *effect test requirement string*. The last two are using values form *{0,1,U}*.

b. The *second* line is also divided into four areas: (1) identification of the *CCF sensitization* applied. *Stuck-at-0* and *stuck-at-1* are abbreviated by *sa0* and *sa1*, respectively; (2) the *running effect*; (3) *forcing-conditions (causes)*; and (3) the *target-effect*. The last two are flagged with an asterisk ("*").

3. *Test suite legend*: Is a quick reminder of what the various values of the test requirements mean. The left part explains the values assigned to *causes*. The right part explains the values assigned to the effects. This legend it follows the *test suite body*.

For an illustration, see the test suite in Appendix A (Section A.6), where the test suite of the TRP is presented. Note that this is a direct copy of the TSUITE LISTING (generated by the TSINCRE subsystem of DINSERT).

4.3.2 Logging Features

As the name implies, the *Logs system* is a system within DINSERT that runs "parallel" to the major subsystem activities and records their activities. The memory locations containing DINSERT data are then stored into a *logs* file (called LOGS LISTING). This is useful because logs provide quick and easy references to the subsystem interface data-exchange during the test requirement generation. Since logs are occasionally seen by the user, their proper representation and formatting is worthwhile. In the next sections, we present the *logs* of individual DINSERT modules (CCNEDIT, CCFGEN and CCSENS; see Figure 4.1 on page 116) dumped in LOGS LISTING.

4.3.2.1 Logs Generated by the CCNEDIT module

The CCNEDIT module (Figure 4.1 on page 116 and/or in Table 4.7 on page 132) gives rise to data about the *GTF* representation of the

```
201 = ( 103 + 104 + 108 )
202 = 106 . 201
203 = 105 . 107
204 = 102'
205 = 106'
206 = 203'
301 = 101 . 204
302 = 101 . 102
303 = 202 . 203
304 = 202 . 206
305 = 201 . 205
```

Figure 4.3: Logs Generated by the CCNEDIT Module. These logs represent the *GTF's* of the *CCN* shown in Figure A.2 on page 180.

gates in *CCN*. These *logs* are useful because they provide a compact representation (*gate types* and their *fanin/fanout* relationships) of the *CCN*. And this allows us to check if the *CCN* have been edited correctly (see Section A.3). On the other hand, this kind of logs contains all the information we need to construct ("graph") a *CCN* (e.g., Figure A.2 on page 180). Figure 4.3, illustrates this kind of *logs*, based on the TRP.

4.3.2.2 Logs Generated by the CCFGEN Module

The subsystem CCFGEN (Figure 4.1 on page 116 and/or in Table 4.7 on page 132) takes as input the *GTF's* of the *CCN* under investigation, and produces as output the *CCF canonical forms* (*PoS's* and *SoP's*). An illustration of this kind of log appears in Appendix A (Section A.6.1).

4.3.2.3 Logs Generated by the CCFSENS Module

The CCFSENS subsystem (Figure 4.1 on page 116 and/or in Table 4.7 on page 132) takes the *PoS canonical forms* and generates as output the *stuck-at-0 compact test library log* and the *stuck-at-0 compact test library log*. These logs record their corresponding *compact test libraries*. And they contain *ternary* values *{0,1,U}* that serve as "bases" for cross references of the *causes* being combined in the *PoS canonical form* for this *effect*. These logs are as follows:

Stuck-at-0 compact test library log: This log is presented in a matrix form. It provides a cross reference between the associated

Table 4.5: Stuck-at-0 Compact Test Library Logs for TRP.

	301	302	303	304	305	
101	1	1	U	U	U	101
102	0	1	U	U	U	102
103	U	U	1	1	1	103
104	U	U	1	1	1	104
105	U	U	1	0	U	105
106	U	U	1	1	0	106
107	U	U	1	0	U	107
108	U	U	1	1	1	108
	301	302	303	304	305	

causes (values: {0,1,U}) with respect to the *PoS canonical forms* so that the corresponding *effect* becomes "*present*".

For example, if a *cause* appears in the complemented form, the value *{0}* is assigned. If it is in the uncomplemented form then *{1}* is assigned; And if neither of the above (i.e., cause does not participate in this *PoS*) then *{U}* is assigned. Note that the *stuck-at-0 compact test library*, is referenced only while it applies to *stuck-at-0 sensitization* (i.e., with respect to *cause product-terms*, Section 3.4.2). Table 4.5 on page 130, illustrates the *stuck-at-0 compact test library log* for the TRP.

Stuck-at-1 compact test library log: This is an inverted version

Table 4.6: Stuck-at-1 Compact Test Library Logs for TRP.

	301	302	303	304	305	
101'	0	0	U	U	U	101'
102'	1	0	U	U	U	102'
103'	U	U	0	0	0	103'
104'	U	U	0	0	0	104'
105'	U	U	0	1	U	105'
106'	U	U	0	0	1	106'
107'	U	U	0	1	U	107'
108'	U	U	0	0	0	108'
	301	302	303	304	305	

(e.g., *0 <- 1* and *1 <- 0*) of the *stuck-at-0 compact test library log* (Table 4.5 on page 130) with respect to *{0,1}* values. *U's* remain the same. Note that the *stuck-at-1 compact test library* is referenced only when it applies to *stuck-at-1 sensitization* (e.g., sensitization with respect to *cause sum-terms*, Section 3.4.1). The following Table 4.6 on page 131, illustrates the *stuck-at-1 compact test library log* for the TRP.

4.3.3 System Interface Design Features

We now list the CMS-files for each subsystem (Figure 4.1 on page 116) as well as the subprograms (all are subroutines) that belong in each such CMS-file. These are shown in Table 4.7.

DINSERT has been developed in the University's research AMDAHL 470/V8 mainframe, under CMS (VM/SP Release 4), using the VS/FORTRAN compiler (Level 1.4.1 - May 1985) on the VM/370 environment. This mainframe offers a very attractive software development environment mainly due to its resource availability, speed and software development tools such as the VS/FORTRAN Interactive Debugger (Release 2.0). This debugger provides both static analysis and dynamic tracing capabilities. Note that VS/FORTRAN is the IBM version of ANSI FORTRAN-77.

FORTRAN-77 is a high-level structured programming language. It is widely used and supported by almost any computer including mini and micro systems. Portability issues for our system have forced us to use only standard FORTRAN-77 programming features. Therefore, porting the software to another system would involve the following activities: (1) the *transfer* of *source-files* to

Table 4.7: DINSERT Source File Hierarchical Structure.

Subsystem:	*CMS-files:*	*Subroutine names:*
CCNEDIT /	A$MAIN /	BLOCK, THESIS (main program)
	B$UTIL /	OPEN, CLOSE, CLEAR, MSG, ERROR, SCREEN, CHRINT, I3TA3, A3TI3, LOGIN, LOGOUT
	C$INIT /	SETINS, GETCIE, STACIE, CNDUMP
	D$ALLOC /	LGATES, LOGIE
	E$CONN /	CONNEC, CHKFDB, CHKFAN, TRANCI, GETREC, DECOMP
CCFGEN /	F$POS /	POSEXP, INVOLU, MORGAN, IDEMPO, PRTEXT
	G$SOP /	SOPEXP, RSPASS, ITECLR, LSTXRS
CCFSENS /	H$SENS /	PTHSEN, RUNCEG, SENPNL, SENTST, SENEFF, TSDRIV, SOPPNL, SOPDES, POSSRT, POSLNG, SOPLNG, TSDMTX, GETTSD
TSINCRE /	I$TS /	GETSID, TSHELP, TSIPNL, TSDEMO, FILTER, DRIVER, TSSAVE, TSDES, PRTTS
TCGEN /	J$TC /	TSCAU, TSEFF, SUBVAL

another system (*file-names* are listed in Table 4.7); and (2) *recompilation* in the new environment (including system libraries etc.). In our University's environment, KERMIT[1] may be used for the transfer of files between various machines. Via network (NETNORTH) files can also be transferred to another computer system.

[1] KERMIT is a *file transfer protocol* now available in various machines including: *Apple, IBM-PC, Dy/4, Rainbow, VAX,* and the *AMDAHL.*

Basically, each subsystem is broken down to a number of subtasks implemented by a subroutine. The subsystem interface and subroutine parameter passing is done by the labelled COMMON. COMMON blocks have permitted data sharing among the program units selectively. And parameters permitted communication of different data objects to program units on different invocation of these units. Scoping rules (e.g., global scope) of the COMMON block and initialization of variables in the block-data are desirable features that add to *maintenability* and *modularity*.

The flexibility in FORTRAN to develop *modules* and *subsystems*, to *compile* and *load* them in *libraries*, and to have the *loader* automatically retrieve *library units* and *link* them into *object-code* of a system, is a powerful abstraction mechanism used extensively in the development of DINSERT. But the *independent compilation* of FORTRAN has two major weakness: (1) it fails to check alignment of variables in COMMON blocks; and (2) it fails to check agreement between the number and types of parameters in subroutine calls.

4.4 Assessment of DINSERT Prototype

DINSERT is a user-friendly menu-driven interactive prototype system implementing our incremental sensitized test design methodology described in Chapter III.

In Sections 4.4.1 and 4.4.2, we assess DINSERT from the user's and system's points of view. Then we discuss some of its limitations and give recommendations for future enhancements.

4.4.1 Assessment from the User's Point of View

Much has been written about the CEG approach [Elme 73,74,75, Myer 79], viewed as a formal language into which a natural language specification is translated. However, the analysis of this graph, yielding a hierarchy of limited entry "Decision Tables" [Myer 79] or the "Test Library Design" [Elme 73], is a manual process. It is difficult and complicated to manage and construct. Both approaches, CEG and ENF (evaluated in Chapter V), are largely heuristic and not suitable for computerization. On the other hand, no automated tools based on either approach, as far as we know, have been reported in the literature.

In addition to its unique representation of specifications in canonical forms and the construction of a high-level test suite, DINSERT has many attractive features. As far as we know our prototype system is the first computerized tool of its kind automating the test requirement generation in a user-friendly and attractive iterative environment. To get an appreciation of DINSERT and its usefulness one should try it or go through the typical session provided in Appendix A (see Sections A.3, A.4 and A.5).

4.4.2 Assessment from the System's Point of View

To run DINSERT, only elementary CMS knowledge is required. Specifically, to be familiar with the *logon* and *logoff* procedure and the use of the "print" and "listfile" commands (to obtain a hard-copy and to query CMS-files).

To run DINSERT: logon to CMS, type DINSERT then press RETURN. However, before running DINSERT (execution of the DINSERT module), the user should make sure that the DINSERT file environment is set properly (e.g., *DINSERT module* and other *support-files* reside on the CMS A-disk). This environment is shown in Figure 4.2 on page 118. Note that the address-space[2] of the DINSERT module includes: (1) DINSERT *object-code*; (2) the necessary *CMS libraries* (VLNKMLIB and VFORTLIB) and VS/FORTRAN I/O utilities; and (3) a user-defined *macro* CLRSCN TXTLIB (defined in Table 4.1 on page 119).

With regard to *modularity*, DINSERT is a FORTRAN-77 application program with several *program units*: (1) *main-program*; (2) *58 subroutines* (enumerated in Table 4.7 on page 133); and (3) *block-data* of *37 labelled COMMON blocks*. The above program resemples about 86 Kbytes; ranging from 0.2 Kbytes to maximum 5 Kbytes per *program unit*.

The implementation of DINSERT requires extensive use of operations on character strings. These operations are mostly *character comparison, extraction* and *concatenation*. The basic units of such string manipulation are CHARACTER*1 and CHARACTER*3. Since DINSERT is meant to be an interactive system, we have sacrificed static storage allocation (FORTRAN) to gain in speed. This means that the string manipulation and data processing have been carried out "*in-core*" (alternatively, "*in-disk*" would be much slower due to disk I/O's and other

---

[2] The collection of programs and data that are accessed in a process forms an *address space* (i.e, standard operating system components, user-specific tasks, etc.).

overheads).

Furthermore, some *program behavior considerations* such as *structured programing, modularity* and *careful organization* of subroutines in "*logical*" *subsystems* (Table 4.7 on page 133) have improved the locality of memory references. So by keeping the "memory-page-swapping" to a minimum, we have improved DINSERT *performance* (data/string manipulation and user-interaction).

4.4.3 Quality of Documentation

For the subroutine organization, structure, programming style, documentation, etc., it is best to browse through the DINSERT listings. With regard to documentation of a subroutine, we found that 10 to 15 lines of comments explaining the *subroutine interface* (forming the "subroutine headings"), would greatly help one to understand the subroutine.

This concept of documentation is illustrated in Figure 4.4 showing the "headings" of an arbitrary subroutine of our system (e.g., Subroutine DRIVER: TSINCRE / I$TS / DRIVER). Similarly, all the subroutines are equipped with this type of documentation. In addition, informative comments are scattered throughout the *body* of a subroutine to explain the logic, subroutine-call conventions, formatting, etc.

```
*******************************
*
*   • Purpose: Monitors the test requirement generation
*              and puts them in Buffer. Displays (TSDES)
*              Buffer, three tests per screen. Validates
*              user's response as to add (TSSAVE) tests
*              in test suite, etc.
*
*   • Features: Involves user-interaction and Disk I/O's.
*
*   • List of Subprogram I/F:
*
*     CALLed by list:   home:      CALLing list:
*
*     TSDEMO          -> DRIVER -> TSIPNL, TSDES, TSSAVE,
*                                  TSHELP
*
*   • List of parameter passing: (i: input; o: output)
*
*      i      i      i      i      i      i
*     {TSLOW, TSHIGH, SENSE, IDEV, SPALTE, CAU(100,20),
*      i             i      i             i
*      EFF(100,10), SA01, CAUDUE(100,20), EFFDUE(100,10)}
*
*******************************

          SUBROUTINE DRIVER

             ‹interface; declarations; body; formats›

          RETURN
          END
```

Figure 4.4: A Typical Subroutine "Headings" Documentation.

4.4.4 Tool Limitations and Recommendations for Enhancements

DINSERT is a direct implementation of our methodology. While the evaluation of the methodology is discussed in Chapter V, in this section we will discuss some of the implementation issues and memory management considerations.

In our prototype, the *CCN* size has been pre-set to allow specification of *99 causes*, *99 intermediate causes* and *99 effects* (illustrated in Appendix A, Section A.3.2). The *fanin* and *fanout* of {AND,OR} gates is set to 10. The array size for the PoS canonical form has *72 locations* to allow a maximum of *4 lines* of *18* CHARACTER*3 units each, on every *PoS* reporting-instance[3] of the *PoS* canonical form. Similarly, the array holding an *SoP* canonical form has *500 locations* of CHARACTER*3 each (note that this array "explodes" much faster due to sum-term *multiplications*). For example, in the TRP (Section A.6.1) due to effect 304, the PoS(304) becomes 23% full (e.g., occupying *17* out of *72 locations* allocated); while the SoP(304) becomes only 8% full (e.g., occupying *41* out of the *500 locations* allocated). However, if any of the above array sizes are exceeded, message No.59 or No.62 (Appendix B) is displayed and execution continues with the next effect (e.g., effect 305).

The "test-suite-window" is tailored to fit in one screen, allowing a maximum display ("window") of *3 test requirements* per screen. It is equipped with "*vertical screen management*" (e.g., *f-fwd*) to scroll vertically in order to display the *next* 3 test requirements, if any. When the last test requirement has been viewed (through the current "window") the next "window" displays test requirements starting from the top of the their listing. However, only the first *20 causes* (e.g., 101 to 120)

---

[3] Is a "*PoS intermediate form*". For example, in Figure 3.8 on page 91 (Section 3.4.1.1) we have four *PoS* reporting-instances: one per *Level* until the last level is reaching (e.g., L_3). Note that in this example, the sizes of the *PoS* reporting-instances are *1 line* long each.

and the first *10 effects* (e.g., 301 to 310) are displayed: a display ratio of 2 to 1. To be able to view beyond that, we need to enhance the test suite menu (see Appendix B, No.15) with two more entries: i.e., *Sc-Scroll-causes* to monitor the display of the *next* 20 causes (e.g., 121 to 140); while another entry i.e., *Se-Scroll-effects* would enable us to monitor the display of the *next* 10 effects (e.g., 311 to 320). For more display flexibility, *Sc* and *Se* should work independently.

Linking -DINSERT to a graphics system would aesthetically enhance the user-interface, making the system more appealing and attractive. At the same time, it would provide a useful "record keeping" by generating a hard-copy of the target *CCN* based on data already available from the CCNEDIT module (e.g., *GTF* representations, Figure 4.3 on page 129).

In general, any enhancement of the methodology (Chapter III) could easily be implemented in DINSERT because of its *modularity, structure, quality of documentation* and because of our *design approach*.

Chapter V
EVALUATION OF THE METHODOLOGY

It is difficult to develop a metric to evaluate our approach. In this chapter we will give an evaluation based on comparisons with other useful black-box testing approaches.

We will use the Text Reformatter Problem (abbreviated TRP) taken from Goodenough and Gerhart's paper as a benchmark for purposes of comparison [GoGe 75]. This is a common practice in the testing literature [Howd 80, AdGr 83, Wals 83, PrUr 84, Prob 85]. We design tests for this problem by using three different black-box testing techniques: (1) running our test design tool, DINSERT, with detailed results appearing in the tutorial session presented in Appendix A; (2) applying Myers' Cause-Effect Graphing heuristics (abbreviated CEG), presented in Section 5.2; and (3) applying Walsh's Equivalent Normal Form (abbreviated ENF), presented in Section 5.3.

5.1 Application of DINSERT to Benchmark Problem

Our benchmark problem is described in Sections A.1 and A.2. The original problem, however, has been modified slightly (e.g., MAXPOS is fixed, rather than input) so that short test requirements can be developed and, thus, be more appropriate for comparison with the other approaches (CEG and ENF). Part of the

problem was also discussed earlier in Section 3.2.1, to illustrate the various aspects of our methodology, while we have devoted Section 4.4 to the assessment of the tool.

The discussion here is based on Appendix A. The good features of our methodology and therefore its implementing prototype become quickly evident of a walkthrough in Appendix A. Its clarity and attractiveness to use are summarized by the following points:

- The user interface is syntax-directed, guides construction;
- The user interface is graphical;
- Local views of subnets are presented immediately;
- User input automatically validated immediately;
- Keystrokes are minimized;
- Canonical forms are generated automatically;
- The user can double-check the *CCN* from canonical forms;
- Test requirements generated automatically;
- Both presence and absence of effects are tested for;
- Redundant test requirements automatically removed;
- Explanation facility to reassure user;
- All commands menu-driven, require simple response only;
- Target effect presence testing is single-effect-at-a-time;
- Target effect absence testing is single-effect-at-a-time;
- The test suite organization is unique, local and global test requirements counters, forcing-condition (causes), target-effect, comments;

- Exiting DINSERT provides the user with automatic save/print of the test requirements.

The test suite (presented in Section A.6) shows *14 test requirements* for the presence of a single effect followed by *8* non-redundant *test requirements* for the absence of a single effect. Thus, the total of *22 test requirements* compose a more comprehensive set of tests for benchmark than in the literature (Section 5.2.2, Table 5.1 on page 152; and Section 5.3.2, Table 5.2 on page 156).

The obvious partition of the *5 effects* into two independent canonical forms (e.g., $\{G_a, G_b\}$), namely, $G_a(101,102)$ - $(301,302)$ and $G_b(103,104,105,106,107,108)$ - $(303,304,305)$ are realized, respectively, by the two *CCN* independent subnetworks shown in Figure A.2 on page 180.

5.2 Comparison with CEG Heuristic

CEG was described in Sections 1.1.3.2 and 2.2.1. Elmendorf originally proposed it in 1973 [Elme 73]. Since then, both [Elme 74,75] and [Myer 76,79, Pres 82, AdGr 83, Wals 83, Prob 85] have furher investigated the approach. CEG is a useful technique for guiding the systematic selection of a high-yield set of test cases.

In addition, insight is gained by converting the system specification into the boolean graph. In the process incompleteness, ambiguities, and inconsistencies in the specification may be discovered. Examples of errors of

incompleteness are: (1) a *cause* or *combination* of *causes* with no corresponding *effect*; or (2) encountering an *effect* with no corresponding *cause* [Myer 76]. Also, it is likely in any graph certain combinations of *causes* and *effects* are impossible because of *syntactical* or *environmental* constraints. Such constraints should be added to the graph to eliminate the impossible test cases discussed in Section 2.2.1.2. Thus, Myers' CEG technique is a useful testing technique.

5.2.1 General Comparison

Compared to our approach, CEG (based on Myers' heuristics) contains some shortcomings. For example, Myers' heuristic is directed towards only those tests which cause an effect to be present. In our approach both the "presence" and/or "absence" of an effect are each investigated separately with the *stuck-at-0* and *stuck-at-1 sensitization* procedure respectively. Another shortcoming of the CEG is that the graph construction is time consuming activity and cannot be entirely automated.

The greatest difficulty, however, in applying the CEG heuristic is the complex way in which it is described. Myers' CEG test design was discussed in Section 2.2.1 [Myer 79]. This procedure is based on three recursive principles (Myers' considerations) illustrated in Figure 2.12 on page 55. Myers' specification of the procedure for generating test requirements is ambiguous [AdGr 83]. Also, as the problem specification becomes more complex, the size of the graph increases and ability to work with it decreases quickly. In this case, when a large

number of potential test requirements are generated, Myers suggests using ranking for each test requirement based on estimates of its detection yield [Myer 76]. However, no suggestion is given as to how to carry out such a ranking scheme.

In the following sections, we give a specific comparison to our approach (e.g., running DINSERT) based on simple CEG subgraphs namely an {OR} gate, and {AND} gate, and the more complex series of gates in Figure 5.1 on page 151 taken from Myers' example [Myer 79, pp.68].

5.2.1.1 CEG OR Consideration

Consider a 3 input {OR} gate of a CEG sub-section e.g., $x = (a+b+c)$. Running DINSERT we obtain:

PoS (301) = 204

= (201 + 202 + 203)

= (101 + 102 + 103)

SoP (301) = 101 +

102 +

103

The generated test requirements in the following test suite are:

TestSuiteID: ‹MYERS' 3-INPUT {OR}-GATE.› of (3:Cau, 4:ICau, 1:Eff)																																
TR#	curTR	C A U S E S																			E F F E C T S										
-	/																															
CCF	totTR	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3
sen	-	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	0	0	0	0	0	0	0	0	0	1
typ	EffID	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	
1	1/ 3	1	0	0																												
sa0	301	*																														
2	2/ 3	0	1	0																												
sa0	301		*																													
3	3/ 3	0	0	1																												
sa0	301			*																												
4	1/ 1	0	0	0																												
sa1	301	*	*	*																												

Note that three *stuck-at-0* test requirements and one *stuck-at-1* test requirement are generated. Our approach generates the same test requirements as Walsh's interpretation of CEG (based on Myers' consideration-one, see Figure 2.12 on page 55). Both exclude the test requirement where all 3 inputs are 1, while both agree to include the situation where all 3 inputs are 0 (e.g., test requirement: No.4). However, Myers' representation of the heuristic is not compatible (this is discussed in Section 5.2.1.3).

Therefore, out of 2^3 possible combinations (based on a 3 input {OR} gate) our approach selects only 4 as high-yield test requirements, namely {(1,0,0), (0,1,0), (0,0,1), (0,0,0)}, as shown in the test suite above.

5.2.1.2 CEG AND Consideration

Let us now continue with an {AND} gate subgraph. Consider a 3 input {AND} gate of a CEG sub-section e.g., $x = a.b.c$. Running DINSERT we now obtain:

PoS (301) = 204

= 201 . 202 . 203

= 101 . 102 . 103

SoP (301) = 101 . 102 . 103

TestSuiteID: ‹MYERS' 3-INPUT {AND}-GATE› of (3:Cau, 4:ICau, 1:Eff)

TR#	curTR /	CAUSES																				EFFECTS									
CCF	totTR	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3
sen		0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	2	0	0	0	0	0	0	0	0	0	1
typ	EffID	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1	1/ 1	1	1	1																		1									
sa0	301	*	*	*																		*									
2	1/ 3	0	1	1																		0									
sa1	301	*																				*									
3	2/ 3	1	0	1																		0									
sa1	301		*																			*									
4	3/ 3	1	1	0																		0									
sa1	301			*																		*									

And so this generates one *stuck-at-0* test requirement and three *stuck-at-1* test requirements. Again, both our approach and Walsh's interpretation of CEG (based on Myers' heuristics) agree to <u>exclude</u> the situation where all 3 inputs are 0, while both agree to <u>include</u> the situation where all 3 inputs are 1 (e.g., test requirement: No.1). Again, Myers' presentation clearly

disagrees (this is discussed in Section 5.2.1.3). Therefore, out of 2^3 possible combinations (based on a 3 input {OR} gate) our approach selects only 4 as high-yield test requirements, namely {(1,1,1), (0,1,1), (1,0,1), (1,1,0)} as shown in the test suite above.

5.2.1.3 Combining the Considerations: Errors and Inefficiencies in the CEG Heuristic

In this section, we try to clarify the proper interpretation of Myers' test design considerations by considering the example shown in Figure 5.1 on page 151 taken from Myers' example [Myer 79, pp.68]. As shown in the figure, five test requirements are generated with CEG Myers' heuristics considerations. Our detailed analysis shows:

1. The definition of the CEG heuristic is ambiguous. In particular, he presents two conflicting versions of the heuristic for tracing back through an {AND} gate whose output should be *0*.
2. Consider the CEG generated test requirements (shown at the bottom of Figure 5.1 on page 151):
   a. First, test requirement No.1 would also be produced by *error guessing*;
   b. Secondly, test requirement No.2 is subsumed by test requirements No.3 and No.4, because setting both *causes* 103 and 104 to *0* can mask out which one is actually behaving as if it was *stuck-at-1*. Therefore, out of these five CEG test requirements, No.1 and

No.2 can easily be removed from this set, since they are inappropriate and low-yield.

c. All these test requirements relate only to input conditions (*causes*) that cause the output state (*effect*) to be *0*.

Our approach generates a more targeted set of effective test requirements than Myers'. The test suite (running DINSERT) shown next, displays a total of five high-yield test requirements. Three of these (e.g., test requirement: No.1, No.2, and No.3) are based on *'stuck-at-1 sensitization* and correspond to No.5, No.3, and No.4 of Myers' CEG test requirements (shown at the bottom of Figure 5.1 on page 151), respectively. The canonical forms running DINSERT are:

PoS (301) - 207
- 205 . 206
- (201 + 202) . 203 . 204
- (101 + 102) . 103 . 104

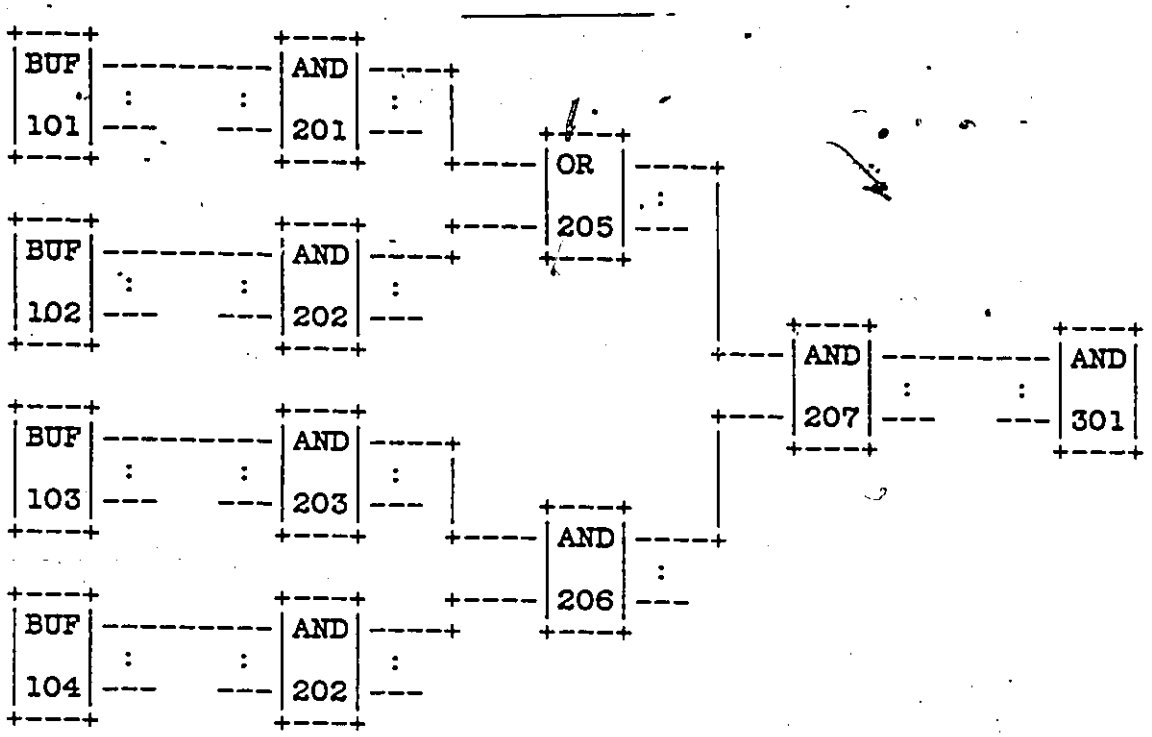
SoP (301) - 103 . 104 . 101 +
103 . 104 . 102

The generated test requirements in the following test suite are:

TestSuiteID: ‹MYERS' TRACING CONSIDERAT› of (4:Cau, 7:ICau, 1:Eff)

TR# / CCF sen typ	curTR / totTR - EffID	CAUSES 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2	EFFECTS 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0 0 1 1 2 3 4 5 6 7 8 9 0
1 sa1	1/ 3 301	0 0 1 1 * *	0 *
2 sa1	2/ 3 301	1 0 0 1 *	0 *
3 sa1	3/ 3 301	1 0 1 0 *	0 *
4 sa0	1/ 2 301	1 0 1 1 * * *	1 *
5 sa0	2/ 2 301	0 1 1 1 * * *	1 *

Note that the first three test requirements relate only to input conditions (*causes*) that cause the output state (*effect*) to be *0*. The remaining <u>two</u> test requirements are based on *stuck-at-0 sensitization* and "complete" our set of test requirements for this example. Our <u>five</u> test requirements, covering both *stuck-at-1* and *stuck-at-0* sensitization, contain more high-yield test requirements.



Effect 301	Causes 101	102	103	104	Test Case No.
0	0	0	0	0	1
0	1	0	0	0	2
0	1	0	0	1	3
0	1	0	1	0	4
0	0	0	1	1	5

Figure 5.1: A Sample CCN for the Illustration and/or Investigation of Myers' Tracing Considerations. This graph (that is the intermediate area) is taken from Myers [Myer 79, pp.67-68].

5.2.2 Comparison with CEG on Specific Benchmark

In this section, we compare the CEG approach (Section 2.2.1) with our approach (Chapter III) with respect to their effectiveness for test requirement generation when applied to the specific benchmark problem (TRP).

Table 5.1: Test Requirements for the TRP Based on the CEG Approach.

		Test Case Number													
		1	2	3	4	5	6	7	8	9	10	11	12	13	14
	101	1	1	-	-	-	-	-	-	-	-	-	-	-	-
C	102	0	1	-	-	-	-	-	-	-	-	-	-	-	-
a	103	-	-	1	0	0	1	0	0	1	0	0	1	0	0
u	104	-	-	0	1	0	0	1	0	0	1	0	0	1	0
s	105	-	-	1	1	1	-	-	-	1	1	1	0	0	0
e	106	-	-	1	1	1	0	0	0	1	1	1	1	1	1
s	107	-	-	1	1	1	-	-	-	0	0	0	1	1	1
	108	-	-	0	0	1	0	0	1	0	0	1	0	0	1
E															
f	301	1	0	0	0	0	0	0	0	0	0	0	0	0	0
f	302	0	1	0	0	0	0	0	0	0	0	0	0	0	0
e	303	0	0	1	1	1	0	0	0	0	0	0	0	0	0
c	304	0	0	0	0	0	0	0	0	1	1	1	1	1	1
t	305	0	0	0	0	0	1	1	1	0	0	0	0	0	0
s															

Applying the CEG approach we obtain the test requirements shown in Table 5.1. The *rows* represent the condition of each *cause* or *effect*, while the *columns* represent a particular *test requirement* to be implemented. Zero ("0") represents the "absent" state and one ("1") the "present" state, while a dash ("-") indicates a "don't care" condition (i.e., their states values are

irrelevant). This decision table contains *14 test requirements*, all of which are covered by those generated by our approach based on *stuck-at-0 sensitization* applied alone.

Our DINSERT test requirements are shown in a test suite (Section A.6). However, note that our test suite is augmented with an additional set of *8 test requirements* based on *stuck-at-1 sensitization* (Section 3.4.2). The total of *22* DINSERT *test requirements* provides a "more complete" set of test requirements. This is because tests are designed to test both "presence" and "absence" of *effects*, and so an additional insight into the dependency of *effects* on *causes* is provided.

5.3 Comparison with Equivalent Normal Form

The other method we will compare to generate a set of test requirements is based on the Equivalent Normal Form (abbreviated ENF) requirements [Arms 66, Koha 70, FrMe 71]. In section 2.1.2.5 and 2.1.3.1 we described and illustrated the ENF approach. The approach we compare here specifically follows Walsh [Wals 83].

5.3.1 General Comparison

Recall that ENF is developed by expressing the output of each gate as a function of inputs and at the same time preserving the identity of each gate. In other words, the ENF of a logic circuit is obtained by exercising (tracing) all paths from the circuit output to every circuit input while, recording the gates through which the path passes.

The major advantage of ENF compared to the CEG approach is that it is more algorithmic and so is simpler to implement. For instance, given the graph generated with CEG, it is straightforward to write a program to implement the ENF algorithm and compute the tables used to generate the test requirements. The ENF-tables are similar to CEG decision tables. One is used to record ENF-related requirements and a second one to record the complement-ENF [Arms 66].

One disadvantage of ENF compared to our approach is that our algorithm for the test requirements is much faster than the algorithm for brute-force path sensitization of ENF. In addition, ENF is charged with an additional overhead of keeping track of the set of paths as well as the trial and error "scoring procedure" [Arms 66]. Moreover, in order to obtain a minimal set of paths, it is generally necessary to check all paths. This has been shown to be impractical for even medium-size graphs [FrMe 71, Even 79, Will 85].

Walsh showed feasibility of using ENF for software testing [Wals 83]. However, the method for generating test requirements is not well explained.

5.3.2 Specific Comparison on Benchmark

We now compare the ENF's effectiveness for test requirement generation versus DINSERT's on the benchmark problem (TRP).

For ENF, we use the cause-effect graph shown in Figure A.2 on page 180. Example ENF's following Walsh's notation are:

$ENF(302) = (101 \text{ AND } 102)_{303}$

$= 101_{302} \text{ AND } 102_{302}$

ENF(305) = $(\text{NOT } 106 \text{ AND } 201)_{305}$

= $(\text{NOT } 106 \text{ AND } (103 \text{ OR } 104 \text{ OR } 108)_{201})_{305}$

= $(\text{NOT } 106_{305} \text{ AND } 103_{201-305})$ OR

$(\text{NOT } 106_{305} \text{ AND } 104_{201-305})$ OR

$(\text{NOT } 106_{305} \text{ AND } 108_{201-305})$.

Recall that each character (e.g., NOT 205_{304}) is called a ***term***, and that terms connected by AND's are called ***literals***. The next step is to test each literal in each ENF for *stuck-at-0*. This is done by assigning 1's to all literals in the term containing it and making all other terms of the current equivalent normal form equal to 0. It is only necessary to test one literal per term (testing more will result in duplicate tests). Applying these procedures the following results are obtained:

For ENF(302) - ***Literal-1***

$101_{302} = 1, \quad 102_{302} = 1$

For ENF(305) - ***Literal-1***

$\text{NOT } 106_{305} = 1, \quad 106_{305} = 0$

$103_{201-305} = 1, \; 104_{201-305} = 0, \; 108_{201-305} = 0$

- ***Literal-2***

$106_{305} = 0$

$103_{201-305} = 0, \; 104_{201-305} = 1, \; 108_{201-305} = 0$

- ***Literal-3***

$106_{305} = 0$

$103_{201-305} = 0, \; 104_{201-305} = 0, \; 108_{201-305} = 1$

Similar expressions can be developed for the remaining ENF's: ENF(301), ENF(303), and ENF(304). These expressions contain assignments to *causes* that cause a sensitized path from *cause* to *effect*, just as in our approach.

Table 5.2: Test Requirements for the TRP Based on the ENF Approach.

Effects	Causes								Test case number
	101	102	103	104	105	106	107	108	
301	1	0	-	-	-	-	-	-	1
302	1	1	-	-	-	-	-	-	2
303	-	-	1	0	1	1	1	0	3
	-	-	0	1	1	1	1	0	4
	-	-	0	0	1	1	1	1	5
304	-	-	1	0	0	1	1	0	6
	-	-	1	0	1	1	0	0	7
	-	-	0	1	0	1	1	0	8
	-	-	0	1	1	1	0	0	9
	-	-	0	0	0	1	1	1	10
	-	-	0	0	1	1	0	1	11
305	-	-	1	-	-	0	-	-	12
	-	-	-	1	-	0	-	-	13
	-	-	-	-	-	0	-	1	14

The test requirements developed by using this method are shown in Table 5.2. These are <u>exactly</u> the same test requirements generated using the CEG approach (shown in Table 5.1 on page 152, Section 5.2.2), as well as these by our DINSERT approach except, for some notation differences (Appendix A, Sections A.6). Table 5.2 shows a disadvantage of ENF reports, because only the target effect

value is indicated. Both CEG and our approach provide information about the other effects per test requirement being generated. Again, note that in the above *14 test requirements*, all are covered by those generated by our approach based on *stuck-at-0 sensitization* applied alone.

The ENF algorithm, however, also requires that test requirements be generated for *stuck-at-1* fault (a false output is expected) for all output conditions. However, this was not required in this case, since Walsh tactfully chose outputs that are mutually exclusive. Again, in DINSERT, we choose to generate these additional test requirements since they provide insight into the dependency of *effects* on *causes* (*8 test requirements*, viewed in our test suite in Section A.6).

5.4 General Observations on Black-Box Test Design Approaches

The final but most difficult step in the testing process is to convert the test requirements into actual test cases. This task, still manual, is carried out in a trial-and-error manner and demands the tester's experience and ingenuity.

"*Deriving test requirements is not the most important part of test design, but is very important for guiding the testing process*". This final step of test implementation is very challenging, but aided significantly by test requirements (now generated automatically by DINSERT).

Table 5.3: An Illustration of a Test Case Derived by its Corresponding Test Requirement

This test case can be used to exercise our sample TRP shown in Figure A.1 on page 177.

Test Case number	Given INPUT conditions	Expected OUTPUT conditions
No.3	A\B#•	A\B
‹etc...›		

Where: \ ‹- BL (Blank character)
• ‹- ET (End-of-Text indicator)
‹- NL (New-Line indicator)

To illustrate the complexity remaining in the testing process we will now show how to implement a test case satisfying one choice of the requirements for test case No.3 and the expected OUTPUT both showing in Table 5.3. The remaining test cases can be implemented in a similar fashion. Note that test requirement No.3 can be viewed either in Table 5.1 on page 152 of CEG, or in Table 5.2 on page 156 of ENF or in the test suite provided by our approach shown in Section A.6.

To summarize, our approach benefits from the CEG approach which offers a rigorous way in transforming a natural language specification into a more formal specification, but with regard to test requirement generation it is closer to the ENF approach. However, both approaches (CEG and ENF) produce a test case design that is more economical than brute force or an ad hoc method. And it has the additional benefit of being able to locate ambiguities in the specification.

People that have used the CEG approach, however, often feel that it is too difficult and time consuming due to manual analysis and construction activities. A major advantage of our approach is the support of a well-designed prototype tool (DINSERT) generating a more complete set of high-yield test requirements. Also, the construction of our *standard CCN* (Section 3.1.2) is somehow simpler compared to the CEG one, because it uses only three gates; namely, {AND,OR,NOT} (see Section 3.1.4).

Therefore, overall, this methodology is competitive and helpful, producing more reliable specifications and, thus, more reliable software.

Chapter VI
SUMMARY AND CONCLUSIONS

6.1 Main Contributions of Thesis

In this thesis, we have presented a new test case design approach based on path sensitization. A brief summary of the main contributions it also appears in Section 1.4. This approach consists mainly of two steps. The first is the construction of a *standard CCN* (cause-effect) representation of the system under test. This step requires both considerable testing experience and ingenuity as well as a thorough study of the specifications. The second step is to use path sensitization techniques to generate a set of high-yield test requirements.

To support this approach, we have designed and developed a prototype system implementing our methodology. This system provides a user-friendly interactive environment, including a helpful editor for constructing the *cause-effect network* and an explanation facility.

Next, we provided a comparison of incremental sensitized test case design methodology with the other two competing black-box testing approaches (ENF and CEG). We noted the following advantages of DINSERT: DINSERT uses *CCN* formalism-precise (introduced in Chapter III), but is accessible to all software engineers. It explores combinations of causes, which become a

unique capability to "cross-exercise" the functional specifications of the target problem (Section 3.5). It gives additional representation of the functional specifications through the *PoS* and *SoP canonical forms* - systematically exploring combinations of specifications (Section 3.4). It does consistency checking because of its redundancy checking capability (Section 3.6). It is an operational support tool, and can incorporate test implementation constraints (e.g., currently, "Exclusive-OR"). It is based on a strong mathematical foundation (combinational logic) using only three types of gates {AND,OR,NOT}. It is very algorithmic. It produces a set of non-redundant high-yield test requirements (see Sections 5.2.1.3, 5.2.2 and 5.3.2). People would prefer to use DINSERT because is clear, systematic, and offers an attractive user-friendly menu-driven interactive environment. These ensure that our approach is superior to other black-box testing techniques.

Finally, the results were given of tests of our prototype system on the well-known benchmark problem for testing techniques the Text Reformatter Problem (abbreviated TRP), presented in detail in Appendix A. This permits specific comparisons of DINSERT with other test design tools (discussed in Sections 5.2 and 5.3). In addition, although not reported here, we have tested DINSERT on a number of other problems such as "*Public Utility Billing System*" [Pres 82], "*CMS Change Subcommand*" [Myer 76], and a variety of multilevel realization logic circuits (e.g., "*Full-Adder*", [Koha 70]). All tests were passed. Its performance has

also been tested under different VM/370 system loads, ranging from 50 to 250 users at a time in the system, with no impact or very little impact on the tool's performance. Space requirements for DINSERT are quite reasonable, namely, about 86 Kbytes (for details refer to Section 4.4.2).

6.2 Limitations and Suggestions for further Research

In practice certain combinations of *causes* are impossible because of semantic or environmental considerations. To account for these, we need to enhance our approach to allow specification of constraints among *causes* as well as *effects*. Using Myers' notation [Myer 79, pp.59], such constraints would include:

1. The *E-constraint* ("exclusive-OR") *E(a,b)*: it requires that at most one of *causes* a, and *b*, can be *1*.
2. The *I-constraint* ("inclusive") *I(a,b,c)*: requires that at least one of *causes a*, *b*, and *c* must always be *1* (*a*, *b*, and *c* cannot be *0* simultaneously).
3. The *O-constraint* ("one-and-only-one") *O(a,b)*: requires that, one and only one, of *causes a* and *b* must be *1*.
4. The *R-constraint* ("requires") *R(a,b)*: requires that whenever *cause a* is *1*, *cause b* must be *1*.
5. The *M-constraint* ("mask") *M(a,b)*: require that, whenever *effect a* is *1*, *effect b* must be *0*.

The *M-constraint* is because there is often a need for a constraint among *effects*. The *M-constraint* makes the independent observation of the *presence* of *effects* possible. The graphical

representation of the above constraints is presented in Section 2.2.1.2 (Figure 2.13 on page 57).

In our prototype system, we have only implemented the *E-constraint* by building "exclusive-OR" characteristics into sensitization of an {OR} gate. However, the modular design of DINSERT allows the straightforward implementation of these constraints. Logically, constraints could be stored (after being specified interactively) as part of the *fanin-list* (for the constraints on causes) and as part of the *fanout-list* (for the constraints on effects) shown currently "empty" in Table 4.3 on page 122. However, a discussion regarding current DINSERT limitations and recommendations for enhancements appears in Section 4.4.4.

The second limitation is in the area of test implementation. It is generally agreed that deriving test requirements is not the most difficult part of test design and development. However, it is very important for guiding the testing process. The most difficult part of test development is the task of implementing the test requirements by choosing appropriate (actual) test data. These test data, in turn, would be used to exercise the system under test. This manual task is carried out by trial-and-error and it is an error-prone activity demanding considerable testing experience and ingenuity.

Progress in automating the actual test-case generation task would greatly advance the testing process. Therefore, our suggestion for further research includes the investigation of

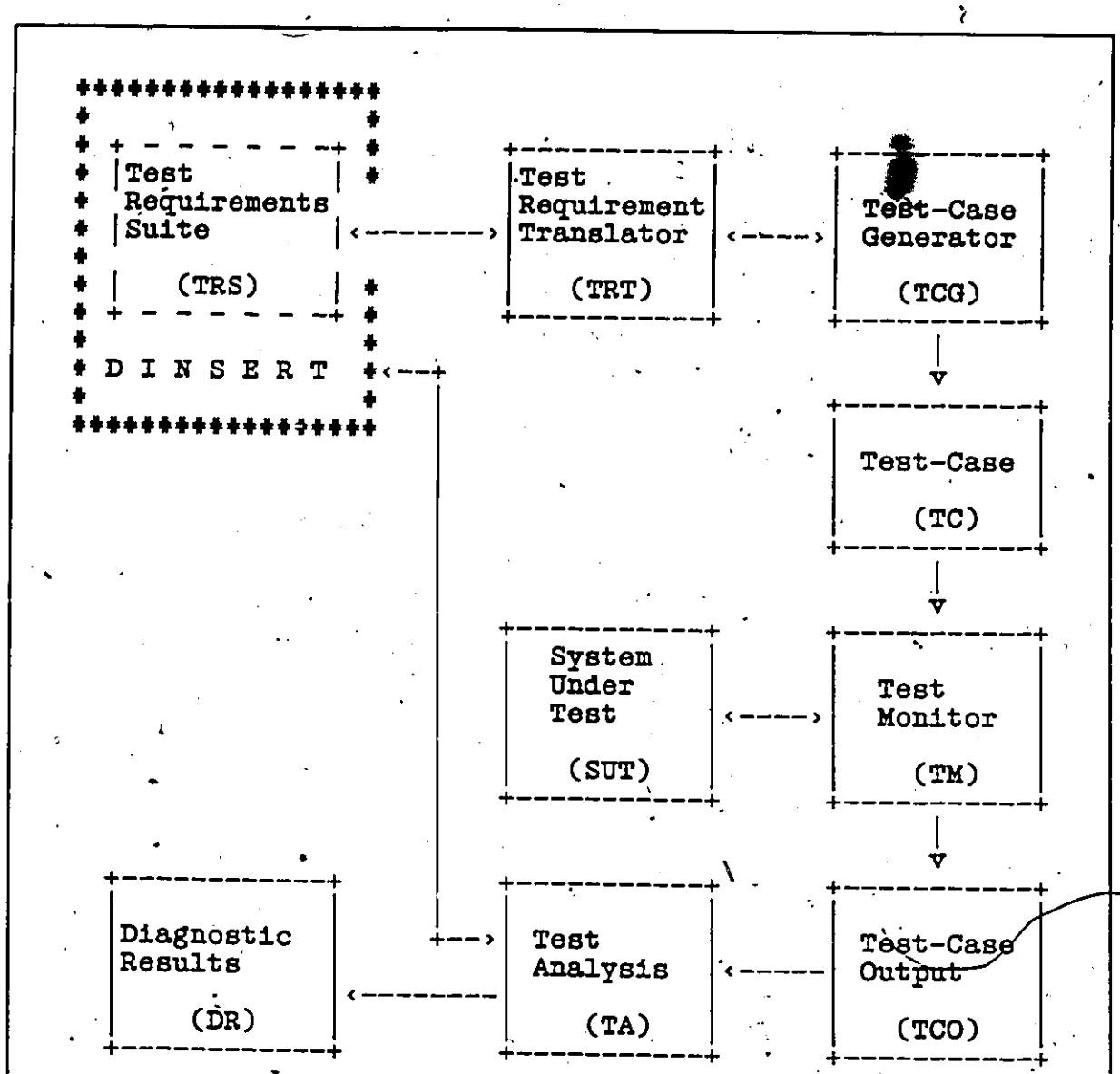


Figure 6.1: A Suggested Architecture of a Future Prototype System and its Subsystem Organization. TRS is the link to DINSERT.

such an undertaking as a natural extension of our approach. A prototype could be developed and linked to our DINSERT. DINSERT already provides a set of non-redundant high-yield test requirements. The new prototype would translate these

requirements into a set of actual test cases which in turn would be used to drive the system under test. Figure 6.1 on page 164 shows the architecture of the suggested prototype system. Its TRT component links to TRS component (or Test-Suite Incrementor of our DINSERT, shown in Figure 4.1 on page 116). Briefly, TCG operates under guidance provided by TRS. TCG in co-operation with TRT translates the strings of *{0,1,U}* and generates corresponding actual test cases (TC component) to test the SUT and store the results in TCO. Note that DR, TA, and TM are responsible for the test analysis and test monitoring, respectively. TCG will be an interactive *test advisor* type of *expert system* that uses rules to derive appropriate *test cases*.

In conclusion, our approach is useful and seems to have considerable potential for cutting down on time and effort in developing appropriate *test suites*. It is directly applicable to high-level testing by exploring effectively the *effects* of *combinations* of *causes* in of functional specifications [PrUr 81].

We believe that this is both an effective and efficient approach for producing more reliable specifications, more effective tests, and therefore, more reliable software.

BIBLIOGRAPHY

[AbSu 85] Abelson, H. and Sussman, G.J. with Sussman, J., "*Structure and Interpretation of Computer Programs*", McGraw-Hill, The MIT Press, 1985.

[AdGr 83] Adler, M. and Gray, M.A., "*A Formalization of Myers Cause-Effect Graphs for Unit Testing*", ACM SIGSOFT, Vol.8, No.5, Oct. 1983, pp.24-32.

[Albe 76] Alberts, D., "*The Economics of Software Quality Assurance*", in National Computer Conference Proceedings, AFIPS, Vol.45, 1976, pp.433-442.

[Agra 72] Agrawal, V.D., and Agrawal, P., "*An Automatic Test Generation System for Illiac IV Logic Boards*", IEEE Trans. Computers, Vol.C-21, No.9, Sept. 1972, pp.1015-1017.

[Appl 83] Appleton, D.S., "*Data-Driven Prototyping*", Datamation, Nov. 1983, pp.259-268.

[Ande 79] Anderson, R.B., "*Proving Programs Correct*", John Wiley, New York, 1979.

[AnBe 81] Andrews, D.M. and Benson, J.P., "*An Automated Program Testing Methodology and its Implementation*", Proc. of the 5th International Conf. Software Engineering, pp.254-261, IEEE Computer Society Press.

[Arms 66] Armstrong, D.B., "*On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets*", IEEE Trans. Electronic Computers Vol.EC-15, No.1, Feb. 1966, pp.63-70.

[Arms 72] Armstrong, D.B., "*A Deductive Method for Simulating Faults in Logic Circuits*", IEEE Trans. Computers Vol.C-21, No.5, May 1972, pp.464-471.

[Beiz 83] Beizer, B., "*Software Testing Techniques*", Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, 1983.

[Beiz 84] Beizer, B., "*Software System Testing and Quality Assurance*", Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, 1984.

[BeLe 72] Belady, L., and Lehman, M., "*An Introduction to Growth Dynamics*", in Statistical Computer Performance Evaluation, Freiberger, W., (ed), Academic Press, 1972, pp.503-511.

[Boeh 76] Boehm, B.W., "*Software Engineering*", IEEE Trans. Computers, Vol.C-25, No.12, Dec. 1976, pp.1226-1241.

[Boeh 83] Boehm, B.W., Gray, T., and Seewaldt, T., "*Prototyping vs Specifyng: A Multi-Project Experiment*", UCLA Computer Science Department Quarterly, 11, pp.93-105.

[BoHo 71] Bossen, D.C. and Hong, S.J., "*Cause-Effect Analysis for Multiple Fault Detection in Combinational Networks*", IEEE Trans. Computers, Vol.C-20, No.11, Nov. 1971, pp.1252-1257.

[Bott 77] Bottorff, P.S., France, S.J., Garges, N.H. and Orosz, E.J., "*Test Generation for Large Logic Networks*", Proc. of the Design Automation Conference, 1977, pp.479-485.

[Boye 75] Boyer R., "*SELECT - A Formal System for Testing and Debugging Programs By Symbolic Execution*", Proc. International Conf. Reliable Software, IEEE Cat. No. 75CH0940-7CSR, 1975, pp.234-245.

[Budd 78] Budd, T.A., et al., "*The Design of a Prototype Mutation System for Program Testing*", in Proceedings of the National Computer Conference, AFIPS, Vol.47, 1978, pp.623-627.

[Budd 80] Budd, T.A., DeMillo, R.A., Lipton, R.J. and Sayward, F.G., "*Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs*", on Conferance Record of the Seventh, Annual ACM Symposium on Principles of Programming Languages, Jan. 1980, pp.220-233.

[Cann 72] Canning, R., "*The Maintenance 'Iceberg'*", EDP Analyzer, Vol.10, No.10, Oct. 1972

[Cava 85] Cavano, J.P., "*Towards High Confidence Software*", IEEE Trans. Software Engineering, Vol.SE-11, No.12, Dec. 1985, pp.1449-1455.

[Chan 65] Chang, H.Y., "*An Algorithm for Selecting an Optimum Set of Diagnostic Tests*", IEEE Trans. Electronic Computers, Vol.EC-14, No.5, Oct.1965, pp.706-711.

[Chan 70] Chang, H.Y., Manning, E.G. and Metze, G., "*Fault Diagnosis in Digital Systems*", Wiley-Interscience, 1970.

[Chap 74] Chappell, S.G., Elmendorf, C.H. and Schmidt, L.D., "*LAMP: Logic-Circuit Simulators*", The Bell System Technical Journal, Vol.53, No.8, Oct.1974, pp.1451-1476.

[ChCh 75] Chang, H.Y. and Chappell, S.G., "*Deductive Techniques for Simulating Logic Circuits*", Bell Telephone Laboratories, Naperville, Illinois, Computer, Mar. 1975, pp.52-59.

[Chea 79] Cheatham T.Jr., at al., "*Symbolic Evaluation and the Analysis of Programs*", IEEE Trans. Software Engineering, Vol.SE-5, No.4, Jul.1979, pp.402-417.

[Chia 72] Chiang, C.L., Reed, I.S. and Banes, A.V., "*Path Sensitization, Partial Boolean Difference, and Automated Fault Diagnosis*", IEEE Trans. Computers, Feb. 1972, pp.189-195.

[Clar 76] Clarke, L., "*A System to Generate Test Data and Symbolically Executed Programs*", IEEE Trans. Software Engineering, Vol.SE-2, No.7, Jun. 1976, pp.215-222.

[Cohe 72] Cohen, A., "*Modular Programs: Defining the Module*", Datamation, Vol.18, Jan. 1972, pp.34-37.

[DeMI 78] DeMillo, R.A., Lipton, R.A. and Sayward, F.G., "*Hints on Test Data Selection: Help for the Practicing Programmer*", Computer, Apr. 1978, pp.34-41.

[Diet 71] Dietmeyer, D.L., "*Logical Design of Digital Systems*", Allyn and Bacon, Boston, 1971. Oct. 1969, pp.284-290.

[Dijk 72] Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., "*Notes on Structured Programming*", Academic Press, New York, 1972, pp.1-82.

[Dijk 77] Dijkstra, E.W., "*The Humble Programmer*", Commun. ACM, Vol.20, No.10, Oct. 1977, pp.858-866.

[Dunn 84] Dunn, R.H., "*Software Defect Removal*", McGraw-Hill, New York, 1984.

[Elme 69] Elmendorf, W.R., "*Controlling the Functional Testing of an Operating system*", IEEE Trans. Systems Science and Sybernetics, Vol.SSC-5, No.4, Oct. 1969, pp.284-290.

[Elme 73] Elmendorf, W.R., "*Cause-Effect Graphs in Functional Testing*", TR-00.2487, IBM Systems Development Division, PoughKeepsie, N.Y., Nov. 1973.

[Elme 74] Elmendorf, W.R., "*Functional Analysis Using Cause-effect Graphs*", IBM Systems Development Division, PoughKeepsie, N.Y. 12602, Presented at Session L201 of SHARE XLIII, Chicago, Illinois, Aug.26th-30th, 1974, pp.566-577.

[Elme 75] Elmendorf, W.R., "*Functional Testing of Software Using Cause-Effect Graphs*", IBM Systems Products Division, PoughKeepsie, N.Y. 12602, Automatic Support Systems Conference (ASSC) for Advanced maintainability, Island Inn, Westbury L.I. New York, Oct.28th-30th, 1975.

[Erco 80] Ercoli, P., "*Decision Tables: Applications to Microprocessors and Some New Implementation Methods*", North-Holland, Euromicro, 1980.

[Even 79] Even, S., "*Graph Algorithms*", Computer Science Press, 1979.

[Fair 84] Fairley, R.E., "*Software Engineering Concepts*", McGraw-Hill Series in Software Engeneering and Technology, 1984.

[Fost 80] Foster, K.A., "*Error Sensitive Test Case Analysis (ESTCA)*", IEEE Trans. Software Engineering, Vol.SE-6, No.6, Jun. 1980, pp.258-264.

[FoOs 76] Fosdick, L.D. and Osterweil L.J., "*Data Flow Analysis in Software Reliability*", ACM Computing Surveys, Vol.8, Sept.1976, pp.305-330.

[Frie 86] Friedman, A.D., "*Fundamentals of Logic Design and Switching Theory*", Computer Science Press, 1986.

[FrMe 71] Friedman, A.D. and Menon, P.R., "*Fault Detection in Degital Circuits*", Prentice-Hall, New Jersey, 1971.

[Gaul 72] Gauld, J.W., Robinson, J.P. and Reddy, S.M., "*Multiple Fault Detection in Combinational Networks*", IEEE Trans. Computers, Vol.C-21, No.1, Jan. 1972, pp.31-36.

[Gale 64] Galey, J.M., Norby, R.E. and Roth, J.P., "*Techniques for the Diagnosis of Switching Circuit Failures*", IEEE Trans. Communication and Electronics, Vol.83, No.74, Sept. 1964. pp.509-514.

[GeYe 76] Gerhart, S. and Yelowitz, L., "*Observations of Falliability in Applications of Modern Programming Methodologies*", IEEE Trans. Software Engineering, Vol.SE-2, No.5, May 1976, pp.195-207.

[Gill 76] Gill, A., "*Applied Algebra for the Computer Science*", Series in Automatic Computation, Prentice-Hall, New Jersey, 1976.

[Goel 85] Goel, A.L., "*Software Reliability Models: Assumptions, Limitations, and Applicability*", IEEE Trans. Software Engineering, Vol.SE-11, No.12, Dec. 1985, pp.1411-1423.

[GoGe 75] Goodenough, J.B. and Gerhart, S.L., "*Towards a Theory of Test Data Selection*", IEEE Trans. Software Engineering, Vol.SE-1, No.2, June 1975, pp.156-173.

[GoMc 80] Goodenough, J.B. and McGowan, C.L., "*Software Quality Assurance: Testing and Validation*", Proceedings of the IEEE, Vol.68, No.9, Sept. 1980, pp.1093-1098.

[Goth 82] Gothmann, W.H., "*Digital Electronics*", Prentice-Hall, Inc., 1982.

[GrCa 71] Griswold, V.M. and Carroll, C.C., "*Large-Scale Circuit Interconnection for Boolean Function Implementation*", IEEE Trans. Computers, May 1971, pp.572-575.

[Hogg 75] Hogger, E.I., "*The Use of Decision Tables in Control Software*", IEE: Trents in On-Line Computer Control Systems, Sheffield, Yorks., England, Apr.21st-24th, 1975, pp.238-245.

[Hopc 73] Hopcroft, J. and Tarjian, R., "*Algorithm 447: Efficient Algorithms for Graph Manipulation [H]*", Commun. ACM, Vol.16, No.6, June 1973, pp.372-378.

[Howd 75] Howden, W.E., "*Methodology for the Generation of Program Test Data*", IEEE Trans. Computers, Vol.C-24, May 1975, pp.554-559.

[Howd 76] Howden, W.E., "*Reliability of the Path Analysis Testing Strategy*", IEEE Trans. Software Engineering, Vol.SE-2, No.9, Sept. 1976, pp.208-215.

[Howd 77] Howden, W.E., "*Symbolic Testing and the DISSECT Symbolic Evaluation System*", IEEE Trans. Software Engineering, Jul.1977, pp.266-278.

[Howd 80] Howden, W.E., "*Functional Testing and Design Abstractions*", The Journal of System and Software 1, 1980, pp.307-313.

[Howd 81] Howden, W.E., "*A Survey of Static Analysis Methods*", Tutorial: Software Testing and Validation Techniques, 2nd Edition, IEEE Computer Society, 1981.

[Howd 82] Howden, W.E., "*Weak Mutation Testing and Completeness of Test Sets*", IEEE Trans. Software Engineering, Vol.SE-8, No.7, July 1982, pp.371-379.

[Hurl 83] Hurley, R.B., "*Decision Tables in Software Engineering*", Van Nostrand Reinhold Data Processing Series, 1983.

[Kaut 68] Kautz, W.H., "*Fault Testing and Diagnosis in Combinational Digital Circiuts*", IEEE Trans. Computers, Apr. 1968, pp.352-366.

[Karp 61] Karp, R.M., McFarlin, F.E., Roth, J.P., Wilts, J.R., "*A Computer Program for the Synthesis of Combinational Switching Circuits*", IBM General Products Division, Development Laboratory, Endicott, N.Y., Proceedings of the Second Annual Symposium and Papers from the First Annual Symposium on Switching Circuit Theory and Logical Design (AIEE), S-124, Sept. 1961, pp.182-195.

[King 76] King, J., "*Symbolic Execution and Program Testing*", Commun. ACM, Vol.19, No.7, Jul. 1976, pp.386-394.

[Klee 67] Kleene, S.C., "*Mathematical Logic*", John Wiley, New York, 1967.

[KoDe 71] Kohavi, Z., and DeWayne, R.P., "*Design Sets of Fault-Detection Tests for Combinational Logic Circuits*", IEEE Trans. Computers, Vol.C-20, No.12, Dec. 1971, pp.845-851.

[Koha 70] Kohavi, Z., "*Switching and Finite Automata Theory*", McGraw-Hill Computer Science Series, 1970.

[Kope 79] Kopetz, H., "*Software Reliability*", Springer-Verlag, New York, 1979, pp.54-62.

[KoSp 71] Kohavi, Z. and Spires, D.A., "*Designing Sets of Fault-Detection Tests for Combinational Logic Circuits*", IEEE Trans. Computers, Vol.C-20, No.12, Dec. 1971, pp.1463-1469.

[Lond 72] London, K.R., "*Decision Tables*", AUERBACH Inc., 1972.

[Mano 84] Mano, M.M., "*Digital Design*", Prentice-Hall, New Jersey, 1984..

[Mill 77] Miller, E.F., "*Program Testing: Art Meets Theory*", Computer, Jul. 1977, pp.42-71.

[Mill 78] Miller, E.F., "*Introduction to Software Testing Technology*", Tutorial: Software Testing and Validation Techniques, 2nd Edition, IEEE Cat. No. EHO 138-8 , 1978, pp.3-14.

[Mill 80] Miller, E.F., "*Program Testing Technology In The 1980's*", Proceedings of the Conference on Computing in the 1980's, pp.72-79.

[Mill 81] Miller, J., "*Stuctured Retrofit*", in Techniques of Program and System Maintenance, Parikh, G. (ed), Winthrop Publishers, 1981.

[McCR 81] McCracken, D., "*Software in the 80s - Perils and Promises*", Computerworld (special edition), Vol.14, No.38, 1980, p.5.

[McCL 56] McCluskey, E.J. Jr., "*Minimization of Boolean Functions*", The Bell System Technical Journal, Nov. 1956, pp.1417-1444.

[McCL 86] McCluskey, E.J. Jr., "*Logic Design Principles*", Prentice-Hall, New Jersey, 1986.

[Muro 79] Muroga, S., "*Logic Design and Switching Theory*", Wiley-Intersience, New York, 1979.

[Muss 80] Musser, D.R., "*Adstract Data Type Specifications in the AFFIRM System*", IEEE Trans. Software Engineering, Vol.SE-6, Jan. 1980, pp.24-32.

[Myer 76] Myers, G.L., "*Software Reliability: Principles and Practices*", Wiley-Interscience, New York, 1976.

[Myer 79] Myers, G.L., "*The Art of Software Testing*", Wiley-Interscience, New York, 1979.

[Naur 69] Naur, P., "*Programming by Action Clusters*", BIT, Vol.9, No.3, 1969, pp.250-258.

[OsFo 76] Osteweil, L.J. and Fosdick L.D., "*DAVE - A Validation Error Detection and Documentation System for Fortran Programs*", Software Practice and Experience, Oct.-Dec. 1976, pp.473-486.

[Paig 75] Paige, M.R., "*Program Graphs, an Algebra, and Their Implication for Programming*", IEEE Trans. Software Engineering, Vol.SE-1, No.3, Sept. 1975, pp.286-291.

[Parn 72] Parnas, D.L., "*On the Criteria to be Used in Decomposing Systems into Modules*", Commun. ACM, Vol.15, No.5, Dec. 1972, pp.1053-1058.

[Parn 79] Parnas, D.L., "*Designing Software for Ease of Extension and Contraction*", IEEE Trans. Software Engineering, Vol.SE-5, No.2, Mar, 1979, pp.128-137.

[Parn 85] Parnas, D.L., "*Sotware Aspects of Strategic Defense Systems*", Commun. ACM, Vol.28, No.12, Dec. 1985, pp.1326-1335.

[Pets 85] Petschenik, N.H., "*Practical Priorities in System Testing*", IEEE Software, Vol.2, No.5, Sept. 1985, pp.18-23.

[Pfau 78] Pfau, P., "*Applied Quality Assurance Methodology*", in Proceedings of the Software Quality and Assurance Workshop, ACM, San Diago, CA, Nov. 1978, pp.1-8.

[Poag 63] Poage, J.F., "*Derivation of Optimum Tests to Detect Faults in Combinational Circuits*", Proc. Symposium on Mathematical Theory of Automata, Polytechnic Institute of Brooklyn, 1963, pp.483-528.

[Poll 68] Pollack, S.L., "*Decision Tables for Computer System Design and Programming*", ACM Professional Development Seminar, 1968.

[Poll 71] Pollack, S.L., Hicks, H.T. and Harrison, W.J., "*Decision Tables: Theory and Practice*", Wiley-Interscience, New York, 1971.

[Pooc 74] Pooch, U.W., "*Translation of Decision Tables*", Computing Surveys, Vol.6, No.2, June 1974, pp.119-150.

[Pres 82] Pressman, R.S., "*Software Engineering: A Practitioner's Approach*", McGraw-Hill, New York, 1982.

[Prob 82a] Probert, R.L., "*Optimal insertion of Software Probes in well Delimited Programs*", IEEE Trans. Software Engineering, Vol.SE-8, No.1, Jan. 1982, pp.34-42.

[Prob 82b] Probert, R.L., "*Gray-Box (Design Based) Testing Techniques*", Proc. of 15th Hawaii Int. Conf. on System Sciences, 1982, pp.94-102.

[Prob 83] Probert, R.L. Scuce, D.R. and Ural, H., "*Specification of Representative Test Cases Using Logic Programming*", Proc. of 16th Hawaii Int. Conf. on System Sciences, 1983, pp.190-196.

[PrUr 82] Probert, R.L. and Ural, H., "*Incremental Improvemant of Specifications by Testing*", Proc. of Workshop on Effectiveness of Testing and Proving Methods, Avolon, CA, May 1982, pp.37-49.

[PrUr 84] Probert, R.L. and Ural, H., "*High-Level Testing and Example-Directed Development of Software Specifications*", The Journal of Systems and Software, Elsevier Science Publishing Co., Inc., 1984, pp.317-325.

[Prob 85] Probert, R.L., Ural, H. and Smith, E., "*CEG Advisor User's Guide*", Department of Computer Science, University of Ottawa, July 1985.

[Reyn 81] Reynolds, J., "*The Craft of Programming*", Prentice-Hall, New Jersey, 1981.

[RoGo 75] Ross, D.T., Goodenough, J.B. and Irvine, C.A., "*Software Engineering: Process, Principles, and Goals*", Computer, Vol.8, No.5, May 1975, pp.312-322.

[RoKa 62] Roth, J.P. and Karp, R.M., "*Minimization Over Boolean Graphs*", IBM Journal, Apr. 1962, pp.227-238.

[Roth 66] Roth, J.P., "*Diagnosis of Automata Failures: A Calculus and a Method*", IBM Journal, July 1966, pp.278-291.

[Roth 67] Roth, J.P., Bouricius, W.G. and Schneider, P.R., "*Programmed algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits*", IEEE Trans. Electronic Computers, Vol.EC-16, No.5, Oct. 1967, pp.567-580.

[Sell 68] Sellers, F.F., Hsiao, M.Y. and Bearnson, L.W., "*Analyzing Errors with the Boolean Difference*", IEEE Trans. Computers, Vol.C-17, No.7, July 1968, pp.676-683.

[ScDi 68] Schneider, P.R. and Dietmayer, D.L., "*An Algorithm for Synthesis of Multiple-Output Combinational Logic*", IEEE Trans. Computers, Vol.C-17, No.2, Feb. 1968, pp.117-128.

[ScMe 72] Schertz, D.R. and Metze, G., "*A New Representation for Faults in Combinational Digital Logic*", IEEE Trans. Computers, Vol.C-21, No.8, Aug. 1972, pp.858-866.

[Sell 68] Sellers, F.F., Hsiao, M.Y. and Bearnson, L.W., "*Analyzing Errors with the Boolean Difference*", IEEE Trans. Computers, Vol.C-17, No.7, July 1968, pp.676-683.

[ShMC 75] Shedletsky, J.J, and McCluskey, E.J., "*The Error Latency of a Fault in a Combinational Digital Circuits*", Proceedings of the Fault-Tolerant Computing Symposium, 1975, pp.210-214.

[Sloa 72] Sloane, N.J.A., "*On Finding the Paths Through a Network*", The Bell System Technical Journal, Vol.51, No.2, Feb. 1972, pp.371-390.

[StFo 75] Stucki, L.S. and Foshee, L., "*New Assertion Concepts for Self-Metric Software Validation*", Proc. 1975 International Conf. Reliable Software, IEEE Cat. No. 75CH0904-7CSR, pp.59-71.

[SuNa 71] Su, S.Y. and Nam, C-W., "*Computer-Aided Synthesis of Multiple-Output Multilevel NAND Networks with Fan-In and Fan-Out Constraints*", IEEE Trans. Computers, Vol.C-20, No.12, Dec. 1971, pp.1445-1455.

[Turn 84] Turner, R., "*Software Engineering Methodology*", Reston Publishing Company, Inc., Virginia, 1984.

[TaOs 80] Taylor, R.N. and Osteweil, L.J, "*Anomaly Detection in Concurrent Software By Static Data Flow Analysis*", IEEE Trans. Software Engineering, May 1980, pp.265-278.

[UlBa 73] Ulrich, E.G. and Baker, T., "*The Concurrent Simulation of Nearly Identical Digital Networks*", GTE Laboratoties, Waltham, Massachusetts 02154, IEEE: The Proceedings of the 10th Digital Automation Conference, 1973, pp.145-150.

[ViRa 84] Vick, C.R. and Ramamoorthy, C.V., "*Handbook of Sortware Engineering*", Van Nostrand Reinhold Electrical/Computer Science and Engineering Series, 1984.

[Wals 83] Walsh, P.J., "*An Analysis of Test Case Selection*", IBM Corporation, General Technology Division, Endicott, New York 13760, Phoenix Conference on Computers and Communications 2nd, 1983.

[WiEi 77] Williams, T.W. and Eichelberger, E.B., "*Random Patterns within a Stuctured Sequential Logic Design*", Proceedings of the International Test Conference, 1977, pp.19-26.

[Will 85] Williamson, S.G., "*Combinatotics for Computer Science*", Computer Science Press, 1985.

[Wort 79] Wortman, D.B., "*On Legality Assertion in Euclid*", IEEE Trans. Software Engineering, Vol. SE-5, Jul. 1979, pp.359-367.

[ZiCo 68] Zissos, D. and Copperwhite, G.W., "*Logical Design Manual*", Sir Isaac Pitman Ltd. London, 1968.

[Ziss 72] Zissos, D., "*Logic Design Algorithms*", Oxford University Press, 1972.

Appendix A
TUTORIAL SESSION: APPLICATION OF DINSERT TO TEXT REFORMATTER PROBLEM

A.1 Problem Description

The sample problem was selected from Goodenough and Gerhart's (originally proposed by [Naur 69]) and has been widely studied in the software testing literature by a number of articles [GoGe 75, Wals 83, PrUr 84]. A listing of a standard program for the Text Reformatter Problem (abbreviated TRP) is shown in Figure A.1 on page 177. It has been modified slightly (i.e., MAXPOS is fixed, rather than input) so that short test data can be developed. Briefly, the input to the program is a sequence of characters having the following properties:

- The input stream characters are classified as ***break*** and ***nonbreak*** characters. A ***break*** character is a *blank* ("NL"), a *new-line* indicator ("NL"), or an *end-of-text* indicator ("ET") character.
- The final character in the text is ET.
- A ***word*** is a nonempty sequence of *nonbreak* characters.
- A ***break*** is a sequence of one or more ***break*** characters.

```
      PROGRAM  TXTREF
C
      INTEGER        MAXPOS, FILL, BUFPOS
      LOGICAL*1      ALARM
      CHARACTER*1    BL, NL, ET, CW, BUFFER(3)
C
      MAXPOS  = 3
      ALARM   = .FALSE.
      FILL    = 0
C
1000  CONTINUE
      IF ((ALARM .EQ. .FALSE.) .OR.
     +    (CW .NE. ET))                                    THEN
          READ (5, 5000)  CW
          IF ((CW .EQ. BL) .OR.
     +        (CW .EQ. NL) .OR.
     +        (CW. EQ .ET))                                THEN
              IF (BUFPOS .NE. 0)                           THEN
                  IF (((FILL+BUFPOS) .LT. MAXPOS) .AND.
     +                (FILL .NE. 0))                       THEN
                      WRITE (6, 5000)  BL
                      FILL  = FILL+1
                  ELSE
                      WRITE (6, 5000)  NL
                      FILL   = 0
                      WRITE (6, 6000)
     +                      (BUFFER(K), K=1,BUFPOS,1)
                      FILL = FILL + BUFPOS
                      BUFPOS = 0
                  ENDIF
              ENDIF
          ELSE
              IF (BUFPOS .EQ. MAXPOS)                      THEN
                  ALARM = .TRUE.
              ELSE
                      BUFPOS  = BUFPOS + 1
                  BUFFER (BUFPOS) = CW
              ENDIF
          ENDIF
          GO TO 1000
      ENDIF
C
5000  FORMAT (A1)
6000  FORMAT (3A1)
      STOP
      END
```

Figure A.1: A FORTRAN-77 Program of Naur's TRP.

In other words the input text can be viewed as a sequence of *words* (i.e., non-empty sequences of *nonbreak* characters) separated by *breaks* (i.e., sequences of one or more break characters) with possible leading and trailing breaks, and ending with an ET character. Most programs are designed to make use of a number of parameters and internal variables such as the following:

- MAXPOS: Maximum number of characters allowed per output line. We will assume throughout that MAXPOS is set to 3 for the purpose generating short, and simple test data.
- FILL: The number of characters printed on the line so far.
- BUFFER: An internal work area (normally an array) for storing a *word* which has not yet been printed.
- BUFPOS: The number of characters in BUFFER. This is set to zero after the *word* in BUFFER is printed.

The program's output should contain the same sequence of *words* as in the input and should satisfy the following *properties*

- A new line should start only between *words* and at the beginning of the output text, if any.
- A *break* in the input is reduced to a single *break* character in the output.
- As many *words* as possible should be placed on each line (i.e., between successive NL characters).
- No line may contain more than MAXPOS characters (*words* and *break* character).

- An oversize *word* (i.e., a *word* containing more than MAXPOS characters) should cause an error-exit from the program. In this case, a variable ALARM is set to the value "TRUE".

A.2 STEP ONE: Definition of the CCN

Specifications for the TRP are discussed in a number of articles [Naur 69, GoGe 75, Wals 83, PrUr 84, Prob 85]. In our approach, we follow the formalization given by Walsh. The natural language

Table A.1: Definition of Causes and Effects of the TRP Organized in DINSERT Notation.

8 Causes: (101,102, ... ,108)

101 :	Character is not (BL or NL or ET)	
102 :	BUFPOS = MAXPOS	(word is too long)
103 :	Character is NL	
104 :	Character is BL	
105 :	(BUFPOS + FILL) < MAXPOS	(word found will fit on current line)
106 :	BUFPOS ≠ 0	(at least one character found and not printed)
107 :	FILL ≠ 0	(a word was already printed on the line)
108 :	Character is ET	

5 Effects: (301,302, ... ,305)

301 :	No OUTPUT	(character put in BUFFER)
302 :	ALARM	(word size too long)
303 :	BL and BUFFER content are printed	
304 :	NL and BUFFER content are printed	
305 :	No OUTPUT	(multiple or preceding *break* character)

assertions in Section A.1 are now represented in terms of *causes* (input conditions or properties) and *effects* (output conditions

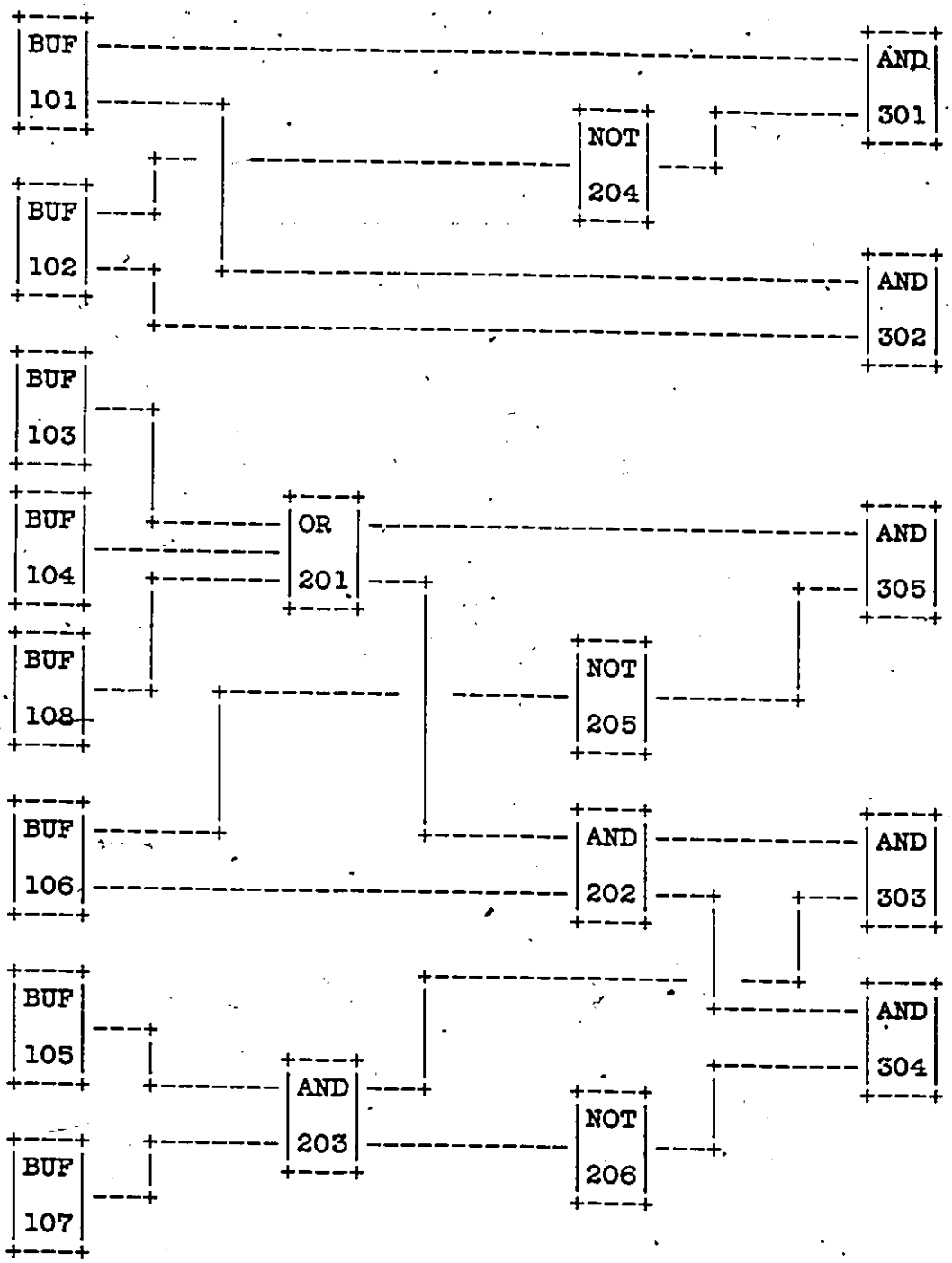


Figure A.2: A Standard CCN Realizing CCF G: $D^8 \dashrightarrow D^5$ of TRP. This *CCN* is based on Table A.1.

or properties), as shown in Table A.1. The semantic content of the TRP is carefully analyzed (shown in Table A.1 on page 179) interconnecting the *8 causes* {101,102, ... ,108} to the *5 effects* {301,302, ... ,305} through *6 intermediate causes* {201,202, ... ,206}. The result of this analysis leads to construction of the *CCN* shown in Figure A.2 on page 180. This *CCN* represents the processing of the next input character in all circumstances.

A.3 STEP TWO: Using DINSERT to Represent the CCN

A.3.1 Entering DINSERT

All the subsystems shown in Figure 4.1 on page 116 are bound together into a large module called DINSERT. To enter DINSERT (also discussed in Section 4.4.2), the user logon to CMS, makes certain that the DINSERT environment is set properly (i.e., Figure 4.2 on page 118). The input files (CCNGDEF, WORKPAD, CLRSCN, DINSERT module) must reside on one of the accessed disks. Then, in CMS type DINSERT followed by ‹CR›. The logo of our system appears on the screen. It is shown in Figure A.3 on page 182.

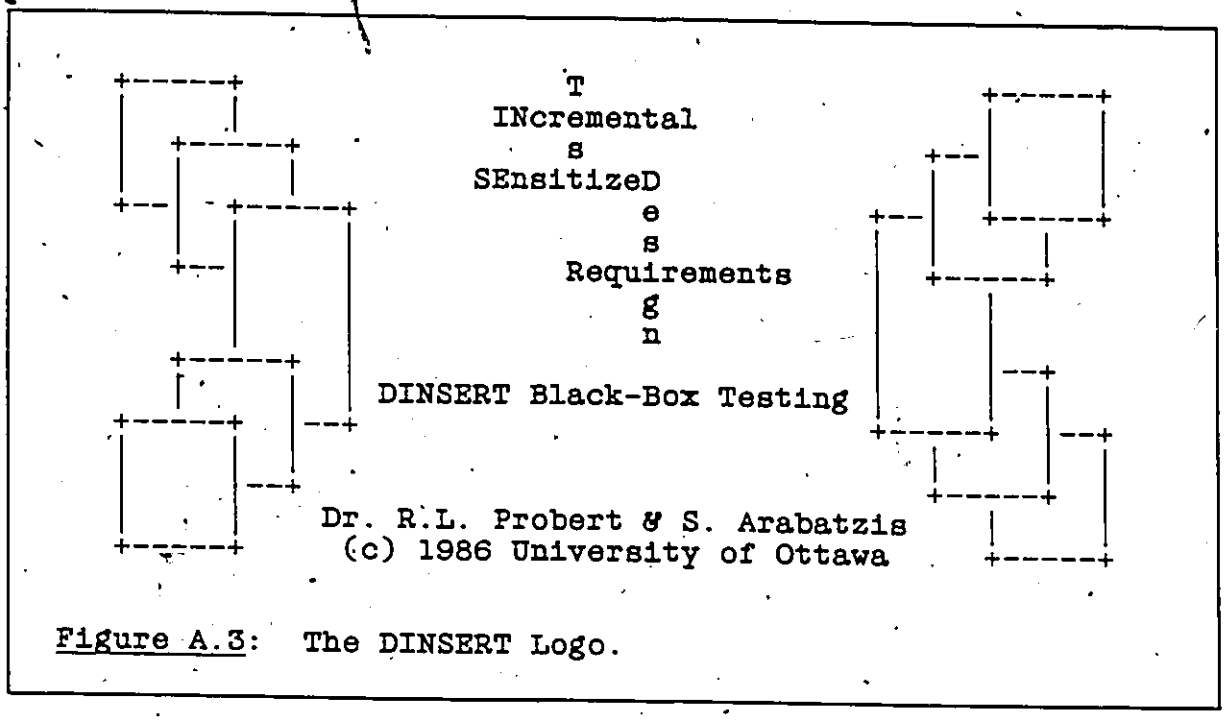


Figure A.3: The DINSERT Logo.

A.3.2 Allocate Storage for the CCN

Through the following *panels* the user is *prompt* to specify the size of the three *CCN* areas (*causes*, *intermediate causes*, *effects*). The responses given are based on the *CCN* in Figure A.2 on page 180. DINSERT *prompts* are preceded by ("--->", see Appendix B). User *responses* are left-justified.

Allocate The Cause-Effect Combinational Network (CCN)			
3 CCN Area Divisions	Initial Area Item Counter	Maximum Area Item Counter	Initially Items/Area
1 CAUSES :	101	199	0
2 INTERM :	201	299	0
3 EFFECTS:	301	399	0

```
---> Enter the No. of "Causes": Maximum is  99
8
---> Enter the No. of "Intermediate Causes": Maximum is  99
6
---> Enter the No. of "Effects": Maximum is  99
5
```

CCN Area Allocation Summary Statistics			
3 CCN Area Divisions	Initial Area Item Counter	Current Area Item Counter	Allocated Items/Area
1 CAUSES :	101	108	8
2 INTERM :	201	206	6
3 EFFECTS:	301	305	5

---› Press ENTER to CLEAR SCREEN and continue:

A.3.3 Assign Gate Types

In the following panels the user is prompted to specify the type of gate for each *intermediate cause* and *effect*. First the *6 intermediate causes* are assigned gate types from {AND,OR,NOT} followed by the *5 effects* from gate types {AND,OR}. There is no need for the user to specify any gate type to causes, since all causes are assigned type {BUF} automatically.

Specify Gates to (6) Corresponding INTERM Area Items			
Area ID	First Item	Last Item	Eligible Gate-Types
INTERM:	201	206	(A=AND, O=OR, N=NOT)

---› (201) first gate
or
---› (202)
and
---› (203)
and
---› (204)
not
---› (205)
not
---› (206) last gate!
not

```
+-------------------------------------------------------------+
|  Specify Gates to ( 5) Corresponding EFFECT Area Items      |
+-----------+-----------+----------+------------------------+
|  Area ID  |First Item |Last Item | Eligible Gate-types    |
+-----------+-----------+----------+------------------------+
|  EFFECTS: |    301    |   305    | (A-AND, O-OR)          |
+-----------+-----------+----------+------------------------+
---> (301) first gate
and
---> (302)
and
---> (303)
and
---> (304)
and
---> (305) last gate !
and
```

A.3.4 Connect Gates Together

The user now enters the last phase of the *CCN* editor to interconnect gates. The system prompts for every gate in the *CCN*.

```
+-------------------------------------------------------------+
|  Connect:                    To:                            |
|                          +---> INTERM : range {201 to 206}  |
|  CAUSE (101,BUF) ---+                                       |
|                          +---> EFFECTS: range {301 to 305}  |
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
301
--->
302
--->
/
+-------------------------------------------------------------+
|  Connect:                    To:                            |
|                          +---> INTERM : range {201 to 206}  |
|  CAUSE (102,BUF) ---+                                       |
|                          +---> EFFECTS: range {301 to 305}  |
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
204
--->
302
--->
/
```

```
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}
|  CAUSE (103,BUF) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
201
--->
/
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}
|  CAUSE (104,BUF) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
201
--->
/
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}
|  CAUSE (105,BUF) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
203
--->
/
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}
|  CAUSE (106,BUF) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
205
--->
202
--->
/
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}
|  CAUSE (107,BUF) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
```

203
---›
/

Connect:	To:
CAUSE (108,BUF) ---+	+---› INTERM : range {201 to 206}
	+---› EFFECTS: range {301 to 305}
Enter / to exit.	

---›
201
---›
/

Connect:	To:
INTERM(201, OR) ---+	+---› INTERM : range {202 to 206}
	+---› EFFECTS: range {301 to 305}
Enter / to exit.	

---›
305
---›
202
---›
/

Connect:	To:
INTERM(202,AND) ---+	+---› INTERM : range {201 to 206}
	+---› EFFECTS: range {301 to 305}
Enter / to exit.	

---›
303
---›
304
---›
/

Connect:	To:
INTERM(203,AND) ---+	+---› INTERM : range {201 to 206}
	+---› EFFECTS: range {301 to 305}
Enter / to exit.	

---›
303
---›
206
---›
/

```
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}|
|  INTERM(204,NOT) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}|
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
301
--->
/
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}|
|  INTERM(205,NOT) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}|
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
305
--->
/
+-------------------------------------------------------------+
|  Connect:                        To:                        |
|                              +---> INTERM : range {201 to 206}|
|  INTERM(206,NOT) ---+        |                             |
|                     +--------+---> EFFECTS: range {301 to 305}|
|  Enter / to exit.                                           |
+-------------------------------------------------------------+
--->
304
--->
/

---> (TestSuiteID: Enter a string up to 25-characters...)
text reformatter problem
```

A.4 STEP THREE: Using DINSERT to Generate Test Requirements

First, the user is asked to select the *sensitization type* (*stuck-at-0* or *stuck-at-1*) and the *effect* to which it should be applied. In general, the order of choosing sensitization type as well as which effect to run is immaterial to the system. However, in this session we choose to apply *stuck-at-0* to all effects (specified in ascending order) first, and then apply *stuck-at-1*.

A.4.1 Stuck-at-0 Sensitization

We now apply *stuck-at-0* sensitization with respect to cause product-terms of effect 301. Note that effects 301 and 302 have quite simple *CCF's*, while effects 303, 304, and 305 are more complex. Through EXPLAIN, the user can view the *CCF's* for the target effect in a variety of three different formats. In this session, due to space problem, we only view some of the *CCF's*. However, options (X and Y) can be seen in LOGS LISTING.

```
+-------------------------------------------------------+
|  CCF Sensitization:                                   |
|                                                       |
|                TESTing: (0=stuck-at-0, 1=stuck-at-1)  |
|  Total of ( 5) EFFECTs:  current range  {301 to 305}  |
|                                                       |
|  Enter / to exit.                                     |
+-------------------------------------------------------+
---> (TEST: select 0/1)
0
---> (EFFECT: select one from the range above)
301
+-----------+----------+-------------------------------+
| TESTing:  |  EFFECT: | (TS /SoP) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-0| (301,AND)| ( 0/ 1) Product-Terms in TS    |
+-----------+----------+-------------------------------+

        1  ->  101 . 102'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
```

A.4.1.1 Stuck-at-0 Based Test Requirement Generation

The test requirement being generated are now displayed through the test requirement reporting facility. This is a "window" of the general test suite (discussed in Section 4.4.2 and Apendix A.6) showing the test requirements generated with respect to currently sensitized *effect*. The various fields of this panel ("test suite window") are self explanatory.

TestSuiteID: «TEXT REFORMATTER PROBLEM.» of (8:Cau, 6:ICau, 5:Eff)

TR#	curTR	C A U S E S	E F F E C T S
-	/		
CCF	totTR	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3 3 3 3 3 3 3 3 3 3
sen	-	0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2	0 0 0 0 0 0 0 0 0 1
typ	EffID	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0
1	1/ 1	1 0 U U U U U U U U	1 0 U U U
sa0	301	* *	*

---» (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
e

Cause TS Legend		Effect TS Legend	
1 :	Invoked	1 :	Present
0 :	Not-Invoked	0 :	Not-Present
U :	Unimportant	U :	Unknown
* :	Forcing-Cond	* :	Target-Effect

---» Press ENTER to CLEAR SCREEN and continue:

```
+-------------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------------+---------------+
|TR#|curTR|           C A U S E S                         |  E F F E C T S|
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen|  -  |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  1|  1/ 1|1 0 U U U U U U                               |1 0 U U U      |
|sa0|  301  |* *                                            |*              |
+---+-----+-----------------------------------------------+---------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
```

Note that at this point *effect* 301 has undergone *stuck-at-0* sensitization and the yielded test requirement been placed in the test suite (Appendix A.6). Similarly, we invoke DINSERT *stuck-at-0* mode on the remaining *effects* {302,303,304,305} and the yielded test requirements are accumulated incrementally in the test suite.

```
+-----------------------------------------------------+
|  CCF Sensitization:                                 |
|                                                     |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)|
|  Total of ( 5) EFFECTs:  current range  {301 to 305}|
|                                                     |
|  Enter / to exit.                                   |
+-----------------------------------------------------+
---> (TEST: select 0/1)
0
---> (EFFECT: select one from the range above)
302
+------------+----------+-----------------------------------+
| TESTing:   |  EFFECT: | (TS /SoP) Summary Statistics      |
+------------+----------+-----------------------------------+
| Stuck-at-0 | (302,AND)| ( 0/ 1) Product-Terms in TS       |
+------------+----------+-----------------------------------+

       1  ->  101 . 102

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
```

```
+-------------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------+-------------------+
|TR#|curTR|           C A U S E S                   |   E F F E C T S   |
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen| -   |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  2|  1/ 1|1 1 U U U U U U                         |0 1 U U U          |
|sa0|  302  |* *                                      |   *               |
+---+-----+-----------------------------------------+-------------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
+-----------------------------------------------------+
|  CCF Sensitization:                                 |
|                                                     |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)|
|  Total of ( 5) EFFECTs:  current range  {301 to 305}|
|                                                     |
|  Enter / to exit.                                   |
+-----------------------------------------------------+
---> (TEST: select 0/1)
0
---> (EFFECT: select one from the range above)
303
```

Based on the selection above, the corresponding *PoS* canonical form of this *effect* (e.g., 303) is "decomposed" into its *product-terms* (e.g., *3 product-terms* are produced) which constitute the basic units of the *stuck-at-0* sensitization. The following panel provides a clear picture of this process.

```
+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /SoP) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-0| (303,AND)| ( 0/ 3) Product-Terms in TS   |
+-----------+----------+-------------------------------+

        1  -> 106 . 105 . 107 . 103

        2  -> 106 . 105 . 107 . 104

        3  -> 106 . 105 . 107 . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
x
```

```
PoS ( 303 )  = 106 .  ( 103 + 104 + 108 )  . 105 . 107

---› Press ENTER to CLEAR SCREEN and continue:

+------------+----------+-------------------------------+
| TESTing:   | EFFECT:  | (TS /SoP) Summary Statistics  |
+------------+----------+-------------------------------+
| Stuck-at-0 | (303,AND)| ( 0/ 3) Product-Terms in TS   |
+------------+----------+-------------------------------+

        1  -›  106 . 105 . 107 . 103

        2  -›  106 . 105 . 107 . 104

        3  -›  106 . 105 . 107 . 108

---› (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
Y

PoS ( 303 )  = 202 . 203

             = 106 . 201 . 105 . 107

             = 106 .  ( 103 + 104 + 108 )  . 105 . 107

---› Press ENTER to CLEAR SCREEN and continue:

+------------+----------+-------------------------------+
| TESTing:   | EFFECT:  | (TS /SoP) Summary Statistics  |
+------------+----------+-------------------------------+
| Stuck-at-0 | (303,AND)| ( 0/ 3) Product-Terms in TS   |
+------------+----------+-------------------------------+

        1  -›  106 . 105 . 107 . 103

        2  -›  106 . 105 . 107 . 104

        3  -›  106 . 105 . 107 . 108

---› (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
z
```

```
SoP ( 303 ) = 106 . 105 . 107 . 103 +
              106 . 105 . 107 . 104 +
              106 . 105 . 107 . 108

---> Press ENTER to CLEAR SCREEN and continue:

+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /SoP) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-0| (303,AND)| ( 0/ 3) Product-Terms in TS   |
+-----------+----------+-------------------------------+

       1  ->  106 . 105 . 107 . 103

       2  ->  106 . 105 . 107 . 104

       3  ->  106 . 105 . 107 . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
2
```

TestSuiteID: ‹TEXT REFORMATTER PROBLEM.› of (8:Cau, 6:ICau, 5:Eff)																																
TR#	curTR	C A U S E S																			E F F E C T S									
-	/																															
CCF	totTR	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3
sen	-	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	2	0	0	0	0	0	0	0	0	0	1
typ	EffID	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
3	2/ 3	U	U	0	1	1	1	1	0																							
sa0	303				*	*	*	*													U	U	1	0	0						
																							*								

```
---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
/
```

```
+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /SoP) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-0| (303,AND)| ( 1/ 3) Product-Terms in TS   |
+-----------+----------+-------------------------------+

        1  -> 106 . 105 . 107 . 103

*A*     2  -> 106 . 105 . 107 . 104

        3  -> 106 . 105 . 107 . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
```

TestSuiteID: ‹TEXT REFORMATTER PROBLEM.› of (8:Cau, 6:ICau, 5:Eff)

TR#	curTR	C A U S E S	E F F E C T S
-	/		
CCF	totTR	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3 3 3 3 3 3 3 3 3 3
sen	-	0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2	0 0 0 0 0 0 0 0 0 1
typ	EffID	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0
3	1/ 3	U U 1 0 1 1 1 0	U U 1 0 0
sa0	303	* * * *	*
4	2/ 3	U U 0 1 1 1 1 0	U U 1 0 0
sa0	303	* * * *	*
5	3/ 3	U U 0 0 1 1 1 1	U U 1 0 0
sa0	303	* * * *	*

```
---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s

+-----------------------------------------------------+
|  CCF Sensitization:                                 |
|                                                     |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)|
|  Total of ( 5) EFFECTs:  current range {301 to 305} |
|                                                     |
|  Enter / to exit.                                   |
+-----------------------------------------------------+
---> (TEST: select 0/1)
0
---> (EFFECT: select one from the range above)
304
```

```
+------------+-----------+-------------------------------+
| TESTing:   | EFFECT:   | (TS /SoP) Summary Statistics  |
+------------+-----------+-------------------------------+
| Stuck-at-0 | (304,AND) | ( 0/ 6) Product-Terms in TS   |
+------------+-----------+-------------------------------+

        1  -> 106 . 103 . 105'

        2  -> 106 . 103 . 107'

        3  -> 106 . 104 . 105'

        4  -> 106 . 104 . 107'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
Y

PoS ( 304 )  - 202 . 206

             - 106 . 201 . 203'

             - 106 .  ( 103 + 104 + 108 )  .  ( 105 . 107 )'

  **MSG** De Morgan's Axiom on: ( 105 . 107 ) '
          (x . y)' -> x' + y'

             - 106 .  ( 103 + 104 + 108 )  .  ( 105'   + 107'   )

---> Press ENTER to CLEAR SCREEN and continue:

+------------+-----------+-------------------------------+
| TESTing:   | EFFECT:   | (TS /SoP) Summary Statistics  |
+------------+-----------+-------------------------------+
| Stuck-at-0 | (304,AND) | ( 0/ 6) Product-Terms in TS   |
+------------+-----------+-------------------------------+

        1  -> 106 . 103 . 105'

        2  -> 106 . 103 . 107'

        3  -> 106 . 104 . 105'

        4  -> 106 . 104 . 107'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
z
```

```
SoP ( 304 )  = 106 . 103 . 105'   +
               106 . 103 . 107'   +
               106 . 104 . 105'   +
               106 . 104 . 107'   +
               106 . 108 . 105'   +
               106 . 108 . 107'

---> Press ENTER to CLEAR SCREEN and continue:

+------------+----------+-------------------------------+
| TESTing:   | EFFECT:  | (TS /SoP) Summary Statistics  |
+------------+----------+-------------------------------+
| Stuck-at-0 | (304,AND)| ( 0/ 6) Product-Terms in TS   |
+------------+----------+-------------------------------+

        1  ->  106 . 103 . 105'

        2  ->  106 . 103 . 107'

        3  ->  106 . 104 . 105'

        4  ->  106 . 104 . 107'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
f
+------------+----------+-------------------------------+
| TESTing:   | EFFECT:  | (TS /SoP) Summary Statistics  |
+------------+----------+-------------------------------+
| Stuck-at-0 | (304,AND)| ( 0/ 6) Product-Terms in TS   |
+------------+----------+-------------------------------+

        5  ->  106 . 108 . 105'

        6  ->  106 . 108 . 107'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
```

```
+-----------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.>, of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------------+-------------+
|TR#|curTR|           C A U S E S                         | E F F E C T S |
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen|  -  |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  9| 4/ 6|U U 0 1 1 1 0 0                               |U U 0 1 0      |
|sa0|  304|       *   * *                                 |       *       |
+---+-----+-----------------------------------------------+---------------+
| 10| 5/ 6|U U 0 0 0 1 1 1                               |U U 0 1 0      |
|sa0|  304|         * *   *                               |       *       |
+---+-----+-----------------------------------------------+---------------+
| 11| 6/ 6|U U 0 0 1 1 0 1                               |U U 0 1 0      |
|sa0|  304|           * * *                               |       *       |
+---+-----+-----------------------------------------------+---------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
f
+-----------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------------+-------------+
|TR#|curTR|           C A U S E S                         | E F F E C T S |
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen|  -  |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  6| 1/ 6|U U 1 0 0 1 1 0                               |U U 0 1 0      |
|sa0|  304|     *   * *                                   |       *       |
+---+-----+-----------------------------------------------+---------------+
|  7| 2/ 6|U U 1 0 1 1 0 0                               |U U 0 1 0      |
|sa0|  304|     *     * *                                 |       *       |
+---+-----+-----------------------------------------------+---------------+
|  8| 3/ 6|U U 0 1 0 1 1 0                               |U U 0 1 0      |
|sa0|  304|       * * *                                   |       *       |
+---+-----+-----------------------------------------------+---------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
+-----------------------------------------------------+
|  CCF Sensitization:                                 |
|                                                     |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)|
|  Total of ( 5) EFFECTs:  current range  {301 to 305}|
|                                                     |
|  Enter / to exit.                                   |
+-----------------------------------------------------+
---> (TEST: select 0/1)
0
---> (EFFECT: select one from the range above)
305
```

```
+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /SoP) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-0| (305,AND)| ( 0/ 3) Product-Terms in TS   |
+-----------+----------+-------------------------------+

        1  ->  106'   . 103

        2  ->  106'   . 104

        3  ->  106'   . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
Y

PoS ( 305 )  = 201 . 205

             =  ( 103 + 104 + 108 )  . 106'

---> Press ENTER to CLEAR SCREEN and continue:

+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /SoP) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-0| (305,AND)| ( 0/ 3) Product-Terms in TS   |
+-----------+----------+-------------------------------+

        1  ->  106'   . 103

        2  ->  106'   . 104

        3  ->  106'   . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
3
```

TestSuiteID: ‹TEXT REFORMATTER PROBLEM.› of (8:Cau, 6:ICau, 5:Eff)

TR#	curTR	C A U S E S	E F F E C T S
-	/		
CCF	totTR	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3 3 3 3 3 3 3 3 3 3
sen	-	0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2	0 0 0 0 0 0 0 0 0 1
typ	EffID	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0
12	3/ 3	U U 0 0 U 0 U 1	U U 0 0 1
sa0	305	* *	*

```
---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
/
```

TESTing:	EFFECT:	(TS /SoP) Summary Statistics
Stuck-at-0	(305,AND)	(1/ 3) Product-Terms in TS

1 -> 106' . 103

2 -> 106' . 104

A 3 -> 106' . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
2

TestSuiteID: ‹TEXT REFORMATTER PROBLEM.› of (8:Cau, 6:ICau, 5:Eff)

TR#	curTR	C A U S E S	E F F E C T S
-	/	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	3 3 3 3 3 3 3 3 3 3
CCF	totTR	0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2	0 0 0 0 0 0 0 0 0 1
sen	-	1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	1 2 3 4 5 6 7 8 9 0
typ	EffID		
12	3/ 3	U U 0 0 U 0 U 1	U U 0 0 1
sa0	305	* *	*
13	2/ 3	U U 0 1 U 0 U 0	U U 0 0 1
sa0	305	* *	*

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
/

TESTing:	EFFECT:	(TS /SoP) Summary Statistics
Stuck-at-0	(305,AND)	(2/ 3) Product-Terms in TS

1 -> 106' . 103

A 2 -> 106' . 104

A 3 -> 106' . 108

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a

TestSuiteID: <TEXT REFORMATTER PROBLEM.> of (8:Cau, 6:ICau, 5:Eff)

TR# - CCF sen typ	curTR / totTR - EffID	C 101	A 102	U 103	S 104	E 105	S 106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	E 301	F 302	F 303	E 304	C 305	T 306	S 307	308	309	310
12 sa0	1/ 3 305	U	U	1*	0	U	0*	U	0													U	U	0	0	1*					
13 sa0	2/ 3 305	U	U	0	1*	U	0*	U	0													U	U	0	0	1*					
14 sa0	3/ 3 305	U	U	0	0	U	0*	U	1*													U	U	0	0	1*					

```
---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
```

A.4.2 Stuck-at-1 Sensitization

We now apply *stuck-at-1* sensitization that is with respect to cause sum-terms starting with effect 301. Note the display of sum-terms based on the SoP canonical form of currently selected effect. Again, through EXPLAIN the user can view the *CCF's* for-- the target effect. Occasionally, will use these options.

```
+-----------------------------------------------------------+
|  CCF Sensitization:                                       |
|                                                           |
|                    TESTing:  (0-stuck-at-0, 1-stuck-at-1) |
|  Total of ( 5) EFFECTs:  current range  {301 to 305}      |
|                                                           |
|  Enter / to exit.                                         |
+-----------------------------------------------------------+
---> (TEST: select 0/1)
1
---> (EFFECT: select one from the range above)
301
```

```
+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /PoS) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-1| (301,AND)| ( 0/ 2) Sum-Terms in TS        |
+-----------+----------+-------------------------------+

        1  ->  101

        2  ->  102'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
x

PoS ( 301 )  = 101 . 102'

---> Press ENTER to CLEAR SCREEN and continue:

+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /PoS) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-1| (301,AND)| ( 0/ 2) Sum-Terms in TS        |
+-----------+----------+-------------------------------+

        1  ->  101

        2  ->  102'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
```

A.4.2.1 Stuck-at-1 Based Test Requirement Generation

The test requirement being generated are now displayed through the test requirement reporting facility. This is a "window" of the general test suite (Appendix A.6).

```
+-------------------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------------------+---------------+
|TR#|curTR|                   C A U S E S                       |  E F F E C T S|
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|3|
|sen| -   |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 15| 1/ 2|0 0 U U U U U U U U U U U U U U U U U U U U U|0 0 U U U      |
|sal| 301  |*                                            |*              |
+---+-----+-----------------------------------------------+---------------+
| 16| 2/ 2|1 1 U U U U U U U U U U U U U U U U U U U U U|0 1 U U U      |
|sal|r301  |  *                                          |*              |
+---+-----+-----------------------------------------------+---------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
```

Note that *effect* 301 has undergone *stuck-at-1* sensitization and the yielded test requirements has been placed in the test suite. Note that test requirement: No. 2/2 is flagged as redundant (denoted by: r301). We now invoke DINSERT in *stuck-at-1* mode to the remaining *effects* {302,303,304,305} to generate and accumulate additional test requirements.

```
+-----------------------------------------------------------+
|  CCF Sensitization:                                       |
|                                                           |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)      |
|  Total of ( 5) EFFECTs:  current range  {301 to 305}      |
|                                                           |
|  Enter / to exit.                                         |
+-----------------------------------------------------------+
---> (TEST: select 0/1)
1
---> (EFFECT: select one from the range above)
302
```

```
+-----------+----------+-----------------------------------+
| TESTing:  | EFFECT:  | (TS /PoS) Summary Statistics      |
+-----------+----------+-----------------------------------+
| Stuck-at-1| (302,AND)| ( 0/ 2) Sum-Terms in TS            |
+-----------+----------+-----------------------------------+

        1  -> 101

        2  -> 102

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
+-------------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------+-----------------+
|TR#|curTR|           C A U S E S                   |   E F F E C T S |
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCP|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen|  -  |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 16| 1/ 2|0 1 U U U U U U                         |0 0 U U U        |
|sal| 302  |*                                        |  *              |
+---+-----+-----------------------------------------+-----------------+
| 17| 2/ 2|1 0 U U U U U U                         |1 0 U U U        |
|sal|r302  |  *                                      |  *              |
+---+-----+-----------------------------------------+-----------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s

+-------------------------------------------------------+
|  CCP Sensitization:                                   |
|                                                       |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)  |
|  Total of ( 5) EFFECTs:  current range  {301 to 305}  |
|                                                       |
|  Enter / to exit.                                     |
+-------------------------------------------------------+
---> (TEST: select 0/1)
1
---> (EFFECT: select one from the range above)
303
```

Based on the selection above, the corresponding *SoP* canonical form of this *effect* (e.g., 303) is "decomposed" into its *sum-terms* (e.g., 4 *sum-terms* are generated) which constitute the basic units (or drivers) of the *stuck-at-1* sensitization. The following panel illustrates this process.

TESTing:	EFFECT:	(TS /PoS) Summary Statistics
Stuck-at-1	(303,AND)	(0/ 4) Sum-Terms in TS

1 -> 106

2 -> (103 + 104 + 108)

3 -> 105

4 -> 107

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a

TestSuiteID: <TEXT REFORMATTER PROBLEM.> of (8:Cau, 6:ICau, 5:Eff)

TR# - CCF sen typ	curTR / totTR - EffID	C A U S E S 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	E F F E C T S 3 3 3 3 3 3 3 3 3 3 0 0 0 0 0 0 0 0 0 1 1 2 3 4 5 6 7 8 9 0
18 sal	2/ 4 303	U U 0 0 1 1 1 0 * * *	U U 0 0 0 *
19 sal	3/ 4 r303	U U 1 0 0 1 1 0 *	U U 0 1 0 *
20 sal	4/ 4 r303	U U 1 0 1 1 0 0 *	U U 0 1 0 *

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
f

```
+-----------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)|
+---+-----+-----------------------------------------+-------------------+
|TR#|curTR|           C A U S E S                   |   E F F E C T S   |
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen|  -  |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 17| 1/ 4|U U 1 0 1 0 1 0                          |U U 0 0 1          |
|sal|  303 |       *                                 |     *             |
+---+-----+-----------------------------------------+-------------------+
| 18| 2/ 4|U U 0 0 1 1 1 0                          |U U 0 0 0          |
|sal|  303 |     * *       *                         |     *             |
+---+-----+-----------------------------------------+-------------------+
| 19| 3/ 4|U U 1 0 0 1 1 0                          |U U 0 1 0          |
|sal|r303 |       *                                 |     *             |
+---+-----+-----------------------------------------+-------------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
+-----------------------------------------------------+
|  CCF Sensitization:                                 |
|                                                     |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)|
|  Total of ( 5) EFFECTs:  current range  {301 to 305}|
|                                                     |
|  Enter / to exit. __                                |
+-----------------------------------------------------+
---> (TEST: select 0/1)
1
---> (EFFECT: select one from the range above)
304

+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /PoS) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-1| (304,AND)| ( 0/ 3) Sum-Terms in TS        |
+-----------+----------+-------------------------------+

      1  ->  106

      2  ->   ( 103 + 104 + 108 )

      3  ->   ( 105'   + 107'   )

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
x

PoS ( 304 )  = 106 .  ( 103 + 104 + 108 )  .  ( 105'   + 107'   )

---> Press ENTER to CLEAR SCREEN and continue:
```

```
+-----------+----------+-------------------------------+
| TESTing:  |  EFFECT: | (TS /PoS) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-1| (304,AND)| ( 0/ 3) Sum-Terms in TS        |
+-----------+----------+-------------------------------+

        1  ->  106

        2  ->   ( 103 + 104 + 108 )

        3  ->   ( 105'   + 107'   )

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a
+-----------------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)        |
+---+-----+-----------------------------------------+-----------------------+
|TR#|curTR|           C  A  U  S  E  S              |   E F F E C T S       |
| - |  /  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|CCF|totTR|1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |1 |
|sen| -   |0 |0 |0 |0 |0 |0 |0 |0 |0 |1 |1 |1 |1 |1 |1 |1 |1 |1 |2 |3 |3 |3 |
|typ|EffID|1 |2 |3 |4 |5 |6 |7 |8 |9 |0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |0 |1 |2 |
+---+-----+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 19| 1/ 3|U  U  1  0  0  0  1  0                   |U  U  0  0  1          |
|sal| 304 |          *                              |          *            |
+---+-----+-----------------------------------------+-----------------------+
| 20| 2/ 3|U  U  0  0  0  1  1  0                   |U  U  0  0  0          |
|sal| 304 |    *  *           *                     |          *            |
+---+-----+-----------------------------------------+-----------------------+
| 21| 3/ 3|U  U  1  0  1  1  1  0                   |U  U  1  0  0          |
|sal|r304 |          *     *                        |          *            |
+---+-----+-----------------------------------------+-----------------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
+-----------------------------------------------------+
|  CCF Sensitization:                                 |
|                                                     |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)|
|  Total of ( 5) EFFECTs:  current range  {301 to 305}|
|                                                     |
|  Enter / to exit.                                   |
+-----------------------------------------------------+
---> (TEST: select 0/1)
1
---> (EFFECT: select one from the range above)
305
```

TESTing:	EFFECT:	(TS /PoS) Summary Statistics
Stuck-at-1	(305,AND)	(0/ 2) Sum-Terms in TS

1 -> (103 + 104 + 108)

2 -> 106'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
x

PoS (305) = (103 + 104 + 108) . 106'

---> Press ENTER to CLEAR SCREEN and continue:

TESTing:	EFFECT:	(TS /PoS) Summary Statistics
Stuck-at-1	(305,AND)	(0/ 2) Sum-Terms in TS

1 -> (103 + 104 + 108)

2 -> 106'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
y

PoS (305) = 201 . 205

= (103 + 104 + 108) . 106'

---> Press ENTER to CLEAR SCREEN and continue:

TESTing:	EFFECT:	(TS /PoS) Summary Statistics
Stuck-at-1	(305,AND)	(0/ 2) Sum-Terms in TS

1 -> (103 + 104 + 108)

2 -> 106'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
z

```
SoP (=305 )  = 106'    . 103 +
               106'    . 104 +
               106'    . 108

---> Press ENTER to CLEAR SCREEN and continue:

+-----------+----------+-------------------------------+
| TESTing:  | EFFECT:  | (TS /PoS) Summary Statistics  |
+-----------+----------+-------------------------------+
| Stuck-at-1| (305,AND)| ( 0/ 2) Sum-Terms in TS        |
+-----------+----------+-------------------------------+

        1  ->   ( 103 + 104 + 108 )

        2  ->  106'

---> (EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP};
      TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
a

+-----------------------------------------------------------------------------+
|TestSuiteID: <TEXT REFORMATTER PROBLEM.> of ( 8:Cau,  6:ICau,  5:Eff)        |
+---+-----+-----------------------------------------+-----------------------+
|TR#|curTR|           C  A  U  S  E  S              |   E F F E C T S       |
| - |  /  +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|CCF|totTR|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|1|3|3|3|3|3|3|3|3|3|3|
|sen| -   |0|0|0|0|0|0|0|0|0|1|1|1|1|1|1|1|1|1|1|2|0|0|0|0|0|0|0|0|0|1|
|typ|EffID|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|1|2|3|4|5|6|7|8|9|0|
+---+-----+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 21| 1/ 2|U U 0 0 U 0 U 0                          |U U 0 0 0              |
|sal| 305  |     * *       *                         |         *             |
+---+-----+-----------------------------------------+-----------------------+
| 22| 2/ 2|U U 1 0 U 1 U 0                          |U U U U 0              |
|sal| 305  |           *                             |         *             |
+---+-----+-----------------------------------------+-----------------------+

---> (S-store-in-TS, E-explain-TS, F-fwd, /-exit)
s
```

A.5 STEP FOUR: Exiting DINSERT

DINSERT allows exit from its environment at all points where the exit-command is indicated (slash: "/").

```
+-------------------------------------------------------+
|  CCF Sensitization:                                   |
|                                                       |
|                TESTing: (0-stuck-at-0, 1-stuck-at-1)  |
|  Total of ( 5) EFFECTs:  current range  {301 to 305}  |
|                                                       |
|  Enter / to exit.                                     |
+-------------------------------------------------------+
---> (TEST: select 0/1)
/
```

When exiting from DINSERT, a message is displayed to remind the user about the existing reporting facilities (files). The file-name and file-type of these files are displayed if a hard-copy is desired. This "exit" message is shown in Figure A.4.

```
* * * * * * *    Exiting DINSERT message    * * * * * * *
*                                                       *
*                                                       *
*   May use system's print command for a hard-copy of   *
*                                                       *
*                                                       *
*   1) Test-suite is stored in file:   TSUITE LISTING   *
*                                                       *
*   2) Generated logs stored in file:  LOGS   LISTING   *
*                                                       *
*                                                       *
*   Note: A file containing this session may be found   *
*         in your virtual reader (fetch and print it)   *
*                                                       *
*                                                       *
* * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

Figure A.4: Exiting DINSERT Message.

A.6 STEP FIVE: Use DINSERT To Document Test Suite Construction

In Section 4.3.1.2, we discussed in detail the organization and the main features of the test suite. Here we give an illustration of the test suite. Our illustration is based on the TRP. The TRP test suite is shown in the following two pages.

Briefly, test requirements in the test suite appear in the order being generated. The order is not important and is user specified. In the running example of this appendix, however, we chose to generate all the *stuck-at-0* based test requirements first. Note that this type of sensitization is with respect to *cause product-terms* (see Section 3.5.1). All the intermediate steps (user-DINSERT interactions), are illustrated throughout Section A.4.1. The *14 test requirements* yield, are shown in the test suite (e.g., *test suite header*: TEXT REFORMATTER PROBLEM) following the *global test requirements counter*: 1 to 14 (see Section 4.3.1.2).

Similarly, the *stuck-at-1* based test requirements follow next. Note that this type of sensitization is with respect to cause sum-terms (see Section 3.5.2). All the intermediate steps (user-DINSERT interactions) are illustrated throughout Section A.4.2 and the *8 test requirements* yield, are shown following the *global test requirements counter*: 15 to 22. So, DINSERT applied to TRP have generated a total ("combination") of *22 test requirements*.

TestSuiteID: <TEXT REFORMATTER PROBLEM.> of (8:Cau, 6:ICau, 5:Eff)

TR# / CCF sen typ	curTR / totTR - EffID	CAUSES 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 / 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0	EFFECTS 3 3 3 3 3 3 3 3 3 3 / 0 0 0 0 0 0 0 0 0 1 / 1 2 3 4 5 6 7 8 9 0
1 sa0	1/ 1 301	1 0 U U U U U U * *	1 0 U U U *
2 sa0	1/ 1 302	1 1 U U U U U U * *	0 1 U U U *
3 sa0	1/ 3 303	U U 1 0 1 1 1 0 * * * *	U U 1 0 0 *
4 sa0	2/ 3 303	U U 0 1 1 1 1 0 * * * *	U U 1 0 0 *
5 sa0	3/ 3 303	U U 0 0 1 1 1 1 * * * *	U U 1 0 0 *
6 sa0	1/ 6 304	U U 1 0 0 1 1 0 * * *	U U 0 1 0 *
7 sa0	2/ 6 304	U U 1 0 1 1 0 0 * * *	U U 0 1 0 *
8 sa0	3/ 6 304	U U 0 1 0 1 1 0 * * *	U U 0 1 0 *
9 sa0	4/ 6 304	U U 0 1 1 1 0 0 * * *	U U 0 1 0 *
10 sa0	5/ 6 304	U U 0 0 0 1 1 1 * * *	U U 0 1 0 *
11 sa0	6/ 6 304	U U 0 0 1 1 0 1 * * *	U U 0 1 0 *
12 sa0	1/ 3 305	U U 1 0 U 0 U 0 * *	U U 0 0 1 *
13 sa0	2/ 3 305	U U 0 1 U 0 U 0 * *	U U 0 0 1 *
14 sa0	3/ 3 305	U U 0 0 U 0 U 1 * *	U U 0 0 1 *

15 sal	1/ 2 301	0 0 U U U U U U *	0 0 U U U *
16 sal	1/ 2 302	0 1 U U U U U U *	0 0 U U U *
17 sal	1/ 4 303	U U 1 0 1 0 1 0 *	U U 0 0 1 *
18 sal	2/ 4 303	U U 0 0 1 1 1 0 * * *	U U 0 0 0 *
19 sal	1/ 3 304	U U 1 0 0 0 1 0 *	U U 0 0 1 *
20 sal	2/ 3 304	U U 0 0 0 1 1 0 * * *	U U 0 0 0 *
21 sal	1/ 2 305	U U 0 0 U 0 U 0 * * *	U U 0 0 0 *
22 sal	2/ 2 305	U U 1 0 U 1 U 0 *	U U U U 0 *

Cause TS Legend		Effect TS Legend	
1 :	Invoked	1 :	Present
0 :	Not-Invoked	0 :	Not-Present
U :	Unimportant	U :	Unknown
* :	Forcing-Cond	* :	Target-Effect

A.6.1 Document the CCF's Constructed by DINSERT

The subsystem CCFGEN (Figure 4.1 on page 116) being a successor of CCNEDIT, takes as input the *GTF's* of the *CCN* under investigation and produces as output the *CCF canonical forms* (i.e., *PoS's* and *SoP's*). The following illustrates this kind of logs, representing the *CCF canonical forms* of the TRP.

PoS (301) = 101 . 204

= 101 . 102'

SoP (301) = 101 . 102'

PoS (302) = 101 . 102

SoP (302) = 101 . 102

PoS (303) = 202 . 203

= 106 . 201 . 105 . 107

= 106 . (103 + 104 + 108) . 105 . 107

SoP (303) = 106 . 105 . 107 . 103 +

106 . 105 . 107 . 104 +

106 . 105 . 107 . 108

PoS (304) = 202 . 206

= 106 . 201 . 203'

= 106 . (103 + 104 + 108) . (105 . 107) '

MSG De Morgan's Axiom on: (105 . 107) '
(x . y)' -> x' + y'

= 106 . (103 + 104 + 108) . (105' + 107')

SoP (304) = 106 . 103 . 105' +
106 . 103 . 107' +
106 . 104 . 105' +
106 . 104 . 107' +
106 . 108 . 105' +
106 . 108 . 107'

PoS (305) = 201 . 205
= (103 + 104 + 108) . 106'

SoP (305) = 106' . 103 +
106' . 104 +
106' . 108

Appendix B
TEXTS OF THE DINSERT MENU COMMAND RESPONSES

In Table 4.4 on page 125 (Section 4.3.1.1) we have classified the DINSERT responses with respect to the subsystem that they are originated. In this Appendix, we provide the listing of the DINSERT responses (a total of 64 responses). For the interpretation of the *response heads* follow the *Legend* provided at the end of this Appendix. The DINSERT responses are:

No:	*Head*:	*Text of DINSERT responses*:
1	---›	Press ENTER to CLEAR SCREEN and continue:
2	---›	Enter the No. of "Causes": Maximum is 99
3	---›	Enter the No. of "Intermediate Causes": Maximum is 99
4	---›	Enter the No. of "Effects": Maximum is 99
5	---›	(nnn) This is the only one gate!
6	---›	(nnn) first gate
7	---›	(nnn) last gate!
8	---›	(nnn)
9	---›	
10	---›	(TestSuiteID: Enter a string up to 25-characters...)
11	---›	(TEST: select 0/1)
12	---›	(EFFECT: select one from the range above)
13	---›	(EFFECT: range contains only one! Enter it: nnn
14	---›	(EXPLAIN:{X-short-PoS, Y-Detailed-PoS, Z-SoP}; TSview: {A-all, enter-a-No.?}; F-fwd; /-exit)
15	---›	(S-store-in-TS, E-explain-TS, F-fwd, /-exit)
16	***›	Invalid input ...
17	***›	Invalid input: Blanks were entered!
18	***›	Still Invalid input: maximum Exceeded!
19	***›	Invalid input: The No.nnn is out of Range!
20	***›	Invalid input: Node No. (nnn) is undefined!
21	***›	Invalid input: EFFECT (nnn) is Undefined or out-of-Range!
22	***›	p/s termNu (nnn) is invalid! Valid panel: {nn to nn} or F-fwd.

```
23  ***> p/s termNu (nnn) is too big! Valid panel: {nn to nn} or F-fwd.
24  ***> p/s termNu (nnn) impossible! Valid panel: {nn to nn} or F-fwd.
25  ***> p/s termNu (nnn) has already been viewed in TestSuite!

26  **MSG** De Morgan's Axiom on: (nnn . nnn ., ... ,. nnn)'
            (x . y)' -> x' + y'
27  **MSG** De Morgan's Axiom on: (nnn + nnn +, ... ,+ nnn)'
            (x + y)' -> x' . y'
28  **MSG** Involution Axiom on: nnn
            (x ' ') -> x
29  **MSG** Idempotent Axiom on: (nnn . nnn) Thus, the above becomes:
            (x . x) -> x
30  **MSG** Idempotent Axiom on: (nnn + nnn) Thus, the above becomes:
            (x + x) -> x

31  ***> INTERM(nnn,AAA) has no FanIn:    This gate is not connected.
         Thus, GTF cannot be generated. Please specify  a connection.

32  ***> EFFECT(nnn,AAA) has no FanIn:    This gate is not connected.
         Thus, GTF cannot be generated. Please specify  a connection.


33  CAUSE (101,BUF) ---> CAUSE (101,BUF): Rejected. Conn. to itself!
34  INTERM(202,AND) ---> INTERM(202,AND): Rejected. Conn. to itself!

35  CAUSE (106,BUF) ---> CAUSE (104,BUF): Rejected. Cascaded CAUSES!

36  INTERM(nnn,AAA) ---> CAUSE (nnn,AAA): Rejected. Tried to Loop-back!
37  INTERM(202,AND) ---> INTERM(201, OR): Rejected. Tried to Loop-back!

38  CAUSE (nnn,AAA) ---> INTERM(nnn,NOT): Rejected. {FanIn Limit - 1}
39  INTERM(nnn,AAA) ---> INTERM(nnn,NOT): Rejected. {FanIn Limit - 1}

40  CAUSE (nnn,BUF) ---> INTERM(nnn,AND): Rejected. {FanOut Limit - 10}
41  CAUSE (nnn,BUF) ---> EFFECT(nnn,AND): Rejected. {FanOut Limit - 10}
42  INTERM(nnn,NOT) ---> INTERM(nnn, OR): Rejected. {FanOut Limit - 10}
43  INTERM(nnn, OR) ---> EFFECT(nnn,AND): Rejected. {FanOut Limit - 10}
44  CAUSE (nnn,AAA) ---> INTERM(nnn,AAA): Rejected. {FanOut Limit - 10}
45  CAUSE (nnn,AAA) ---> EFFECT(nnn,AAA): Rejected. {FanOut Limit - 10}
46  INTERM(nnn,AND) ---> INTERM(nnn,AAA): Rejected. {FanOut Limit - 10}
47  INTERM(nnn, OR) ---> EFFECT(nnn,AAA): Rejected. {FanOut Limit - 10}

48  CAUSE (nnn,AAA) ---> INTERM(nnn,AAA): Already Connected!
49  CAUSE (nnn,AAA) ---> EFFECT(nnn,AAA): Already Connected!
50  INTERM(nnn,AAA) ---> INTERM(nnn,AAA): Already Connected!
51  INTERM(nnn,AAA) ---> EFFECT(nnn,AAA): Already Connected!

52  CAUSE (nnn,BUF) ---> INTERM(nnn,NOT): Already Conn. Unique Conn.

53  CAUSE (nnn,AAA) ---> INTERM(nnn,AAA): Last Valid Connection!
54  CAUSE (nnn,AAA) ---> EFFECT(nnn,AAA): Last Valid Connection!
55  INTERM(nnn,AAA) ---> EFFECT(nnn,AAA): Last Valid Connection!
```

56 +++› PoS(nnn) generation aborted in Sub.POSEXP.
Please exit and Check the CCN and LOGS.

57 +++› Pos(nnn) generation aborted in Sub.MORGAN.
Please exit and Check the CCN and LOGS.

58 +++› PoS(nnn) generation aborted in Sub.IDEMPO.
Please exit and Check the CCN and LOGS.

59 +++› PoS(nnn) generation aborted in Sub.POSEXP. Array PS is full.
Please exit and define a smaller subnetwork for (nnn).

60 +++› CCP's for the effect (nnn) are incorrect. From sub.PRTEXP.
Please exit and check CCP's for (nnn) in LOGS.

61 +++› SoP(nnn) cannot be generated because PoS(nnn) is incorrect.
From sub.SOPEXP. Please exit and check these CCP's in LOGS.

62 +++› SoP(nnn) generation aborted in Sub.SOPEXP. Array SP is full.
Please exit and define a smaller subnetwork for (nnn).

63 +++› PoS(nnn) decomposition to cause sum-terms cannot occur.
No stuck-at-1 test requirements are generated. Continue
with the next effect or exit and check PoS(nnn) in LOGS.

64 +++› SoP(nnn) decomposition to cause product-terms cannot occur.
No stuck-at-0 test requirements are generated. Continue
with the next effect or exit and check SoP(nnn) in LOGS.

Legend to *response heads*:

Prompt	*Warning*	*Axiom MSG*	*Error*	*Gate Conn*
---›	***›	**MSG**	+++›	---›

Appendix C
ILLUSTRATING HARDWARE PATH SENSITIZATION TESTING TECHNIQUES

C.1 The D-Algorithm for Deriving Tests

The algorithm described in Section 2.1.2.4 for deriving tests involve *intersections* of the primitive D-cubes of the fault with propagation D-cubes. While intersecting two D-cubes, $a = (a_1, a_2, \ldots, a_n)$ and $b = (b_1, b_2, \ldots, b_n)$, it is important to remember that all D's in a propagation D-cube may be 0 or 1 and that D''s will always have the opposite value. Thus, if we complement all D's and D''s in a propagation D-cube, the set of vertices they represent do not change. The *intersection* of a D-cube *a* and a *propagation* D-cube *b* is undefined unless:

1. In all positions in which *b* has a D(D'), *a* has a D(D'), or in all positions in which *b* has a D(D') *a* has a D'(D). And,
2. In any position in which *a* has a D or D', *b* does not have a 0 or 1.

If these two conditions are satisfied, then (*a intersection b*) agrees with *a* in all coordinates in which *a* has a D or D'. The remaining coordinates of the intersection are defined as in the intersection of cubes contained in singular covers [Roth 67].

EXAMPLE: As examples, consider *a* = 11-DOD' and *b* = -10D'-D. Then (*a intersection b*) = 110DOD'. If *c* = -1-D'0D', (*a intersection b*) -- = *Q*. Where *a*, *b*, and *c* as in Figure 2.6 on page 34, and "-" denotes "don't-care" values.

C.2 Procedure for Deriving Tests

The procedure for deriving a test for a given fault consists of two parts:

1. A primitive D-cube of the fault is chosen and intersected successively with propagation D-cubes of the blocks of the circuit in order to form a connected chain of D-coordinates to an output. This procedure is called the *D-drive*.
2. The D-cube obtained in (1.) is *intersected* with the singular cover of blocks of the logic circuit until a sufficient number of inputs have been specified. This is called the *consistency* operation.

These two operations correspond to the sensitizing of the path from the fault to an output and the tracing back from the gates in the path towards the inputs in order to specify a sufficient number of inputs to produce the desired internal signals.

The first step in deriving tests is to obtain the *singular covers* and the *propagation D-cubes* of the blocks of the circuit. Only single-input propagation D-cubes are computed initially. Multiple-input D-cubes in which more than one input is D or D' are computed as necessary. These will be necessary when more than one path in a circuit containing reconvergent[1] fanout has to be

[1] If there are two or more paths from any fanout point in a

sensitized in order to detect a fault. Before attempting the D-drive, the blocks of the circuit are ordered so that every block appears after the blocks to whose outputs it is connected.

Any D-cube which represents a partially formed test during the D-drive is called a *test cube* and is represented by *tc* and a superscript denoting the step at which it is obtained. Associated with each test cube is an *activity vector*, consisting of the circuit wires to which D or D' has been propagated at this stage in the test propagation. Thus, the activity vector gives the successors through which the D-chain already constructed may be extended.

The D-drive using single-input propagation D-cubes may terminate prematurely before reaching an output terminal if the circuit contains reconvergent fanout. This will happen when all intersections of the test cube with the propagation D-cubes of all blocks of its activity vector empty. Two types of empty intersections may be encountered. In the first type, both the test cube and the D-cube have opposite values 0 or 1 specified for some coordinate. This happens when conditions necessary for propagation in the earlier part of the D-chain prevent further propagation. In this case the particular D-chain has to be abandoned and a new D-chain using a different member of the activity vector at the step of the last arbitrary selection is attempted. In the second type, the intersection is empty because the propagation D-cube has a 0 or 1 in a coordinate assigned D or

---

circuit to the inputs of some gate, the fanout is said to be reconvergent.

D' in the test cube. This implies that another path in the circuit has been accidently *sensitized*. The D-drive can be continued only if a suitable multiple-input D-cube can be derived for the particular block.

This corresponds to the case where a particular path can be sensitized only alone with other paths, but not by itself. Multiple-input D-cubes with D's or D''s in the desired coordinate positions can be derived in a manner analogous to that the single input D-cubes. If the intersections with the appropriate multiple D-cubes are also empty, we have to back-track to the last step where a choice of path was and try a different path. More details can be found in [Roth 66,67, FrMe 71].

C.3 Examples Based on the Informal Version of the D-Algorithm

It can be shown that the D-algorithm will yield a test for a fault if a test exists [Roth 66]. The algorithm as described above can be automated. For relatively small circuits, the informal method discussed earlier in Section 2.1.3.4, which is a path-sensitizing technique, preserving the identity of signals sensitive to the faults, seems to be more convenient, especially for manual computations.

The D-algorithm can be extended in a rather straightforward manner, to make it applicable to multiple *stuck type faults*. However, we shall consider only an informal version of the algorithm [Roth 67].

The first step in the informal D-algorithm for multiple faults is to label wires which are *stuck-at-0* with D and those which are *stuck-at-1* with D'. As in the single-fault cases, we attempt to propagate a D or D' to the circuit output. In doing so, the effect of a fault may propagate to some other faulty wire. If $x \dashrightarrow y$, indicates that a signal x reaches a faulty wire whose state is represented by y, where $x = 0, 1, D$, or D' and $y = D$ or D', the following addition rules may be defined:

1. D --> D = D
2. D' --> D' = D'
3. D --> D' = 1
4. D' --> D = 0
5. 1 --> D = D
6. 1 --> D' = 1
7. 0 --> D = 0
8. 0 --> D' = D'

These rules, in effect, represent the *consistency* operation in the presence of multiple faults. The validity of the above equations will be clear if we use our interpretation of D and D'. D --> D means that a signal that is normally 1 but changes to 0 in the presence of some fault propagates to a wire which is *stuck-at-0*. The signal on the wire will be 1 if neither fault is present but will be 0 if both faults are present. Similarly D --> D' implies that a signal that changes from 1 to 0 propagates to a wire which is *stuck-at-1*. The signal on the wire will be 1 both in the normal and faulty circuits. The remaining equations can be verified in the same manner.

Figure C.1, shows a circuit with *w stuck-at-0* and *x stuck-at-1*. The signals shown on the wires are obtained by representing the two faults by D and D' respectively and propagating a D to the output.

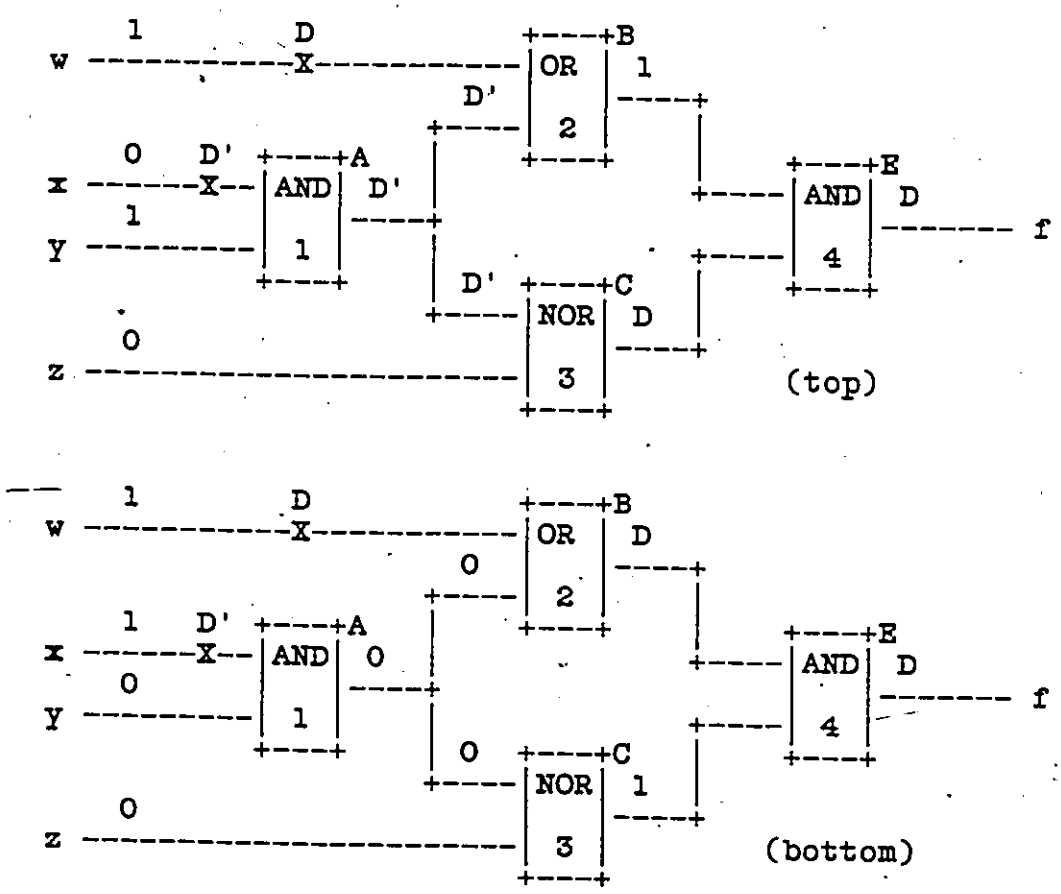


Figure C.1: Informal Version of D-Algorithm Applied to Single-Faults. This figure shows a circuit with *w stuck-at-0* and *x stuck-at-1*.

First let us try to propagate the fault *x stuck-at-1* to the output in the presence of *w stuck-at-0*. To propagate the fault to the output gate A (number 1) we set $x = 0$, $y = 1$. At this state

we have a choice as to paths. Choosing path ACE we must set $z = 0$ resulting in D as output from gate C (number 3). If $w = 0$ the output of B is D', the output of E is D.D' = 0 and the fault is not detected. However, if $w = 1$ the inputs to gate B are D and D', the output is 1 and the output of gate E is D. This test, $w.x'.y.z'$, is shown in the top part of Figure C.1 on page 223.

It is also possible to propagate the fault *w stuck-at-0*, by the input combination $w.y'.z'$, which is shown in the top part of Figure C.1 on page 223 Note that the output of the first AND gate (number 1) will be 0 whether or not *x* is *stuck-at-1*, because $y = 0$. The tests derived by this procedure are guaranteed to detect the simultaneous presence of all the multiple faults considered. Individual faults may or may not be detected by these tests. For example, $w.x'.y.z'$ will detect *w stuck-at-0*, *x stuck-at-1* or both whereas $w.y'.z'$ will detect *w stuck-at-0* or the double fault, but not *x stuck-at-1*.

EXAMPLE: Consider the same circuit repeated in Figure C.2 on page 225, with *x stuck-at-0* and *a stuck-at-1*. In the top part of Figure C.2 on page 225, we attempt to propagate the D on *x* to the output of date E (number 4) the path ACE. If $w = 0$, the output of B is D. Gate E produces a D' output with *a stuck-at-1* because D --> D' = 1. If $w = 1$, the output of B is 1 and 1 --> D' = 1 causing the output of E to be D', as shown in bottom part of Figure C.2 on page 225. Both the tests $w'.x.y.z'$ and $w.x.y.z'$ detect the simultaneous presence of *x stuck-at-0* and *a stuck-at-0*, and also the single-fault *x stuck-at-0*, but not the single-

fault *a stuck-at-1*. The multiple-fault and the single-fault *a stuck-at-1* can also be detected by $w'.y'.z'$, which is derived by setting $y = 0$ so as not to propagate the fault on x and then propagating the fault on a.

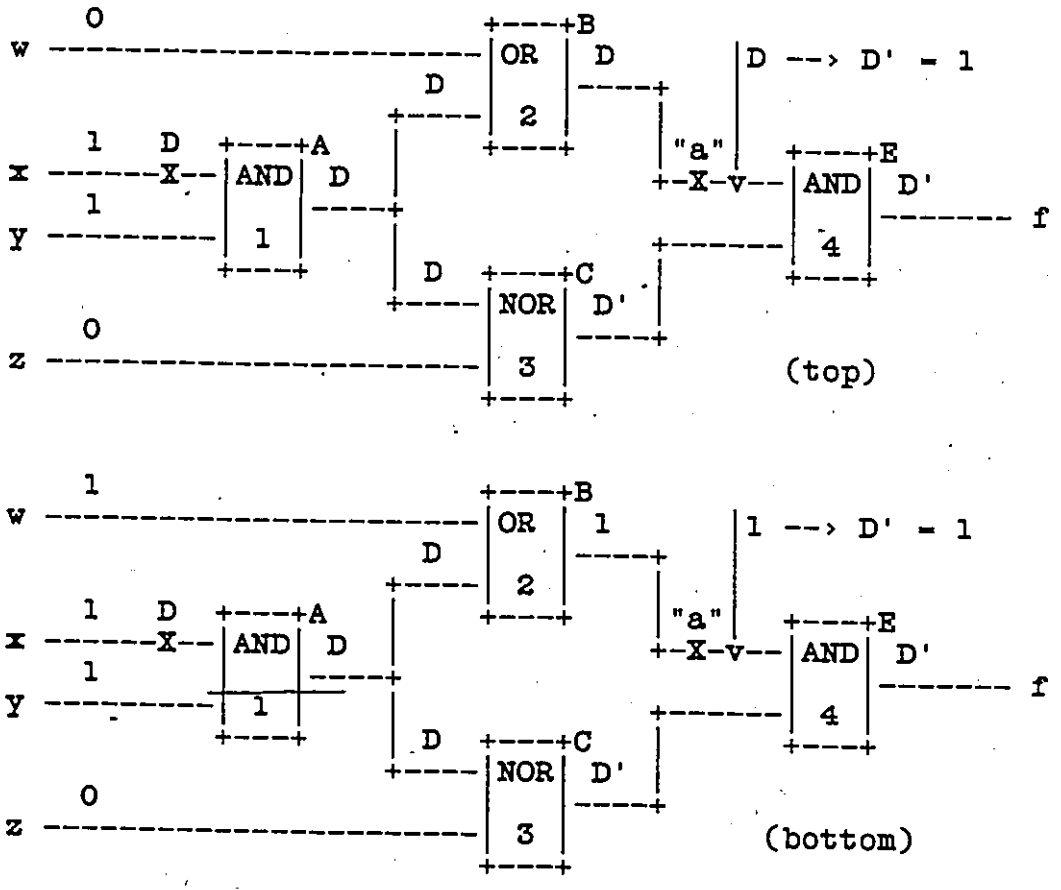


Figure C.2: Informal Version of D-Algorithm Applied to Multiple-Faults. This figure shows a circuit with *x stuck-at-0* and *a stuck-at-1*.