AD-A054 910 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 14/2
AUTOMATIC TEST PROGRAM GENERATION.(U)
MAR 78 Y K CHANG DAAA25-75-C-0650
UNCLASSIFIED 77-02 ECOM-75-0650-F-1 NL

1 OF 4
AD
A054910

(12) Sc

Research and Development Technical Report

ECOM - 75-0650-F-1

AUTOMATIC TEST PROGRAM GENERATION

Yung Ko Chang

Moore School of Electrical Engineering
Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA 19104

March 1978

Final Report for Period 30 June 1975 - 31 August 1977

Prepared for:

ECOM

US ARMY ELECTRONICS COMMAND FORT MONMOUTH NEW JERSEY 07703

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

(18) (19) REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ECOM-75-0650-F-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* (6) AUTOMATIC TEST PROGRAM GENERATION		5. TYPE OF REPORT & PERIOD COVERED (9) FINAL Report 30 Jun 75 — 31 Aug 77
		6. PERFORMING ORG. REPORT NUMBER (14) 77-02
7. AUTHOR(s) (10) Yung Ko Chang		8. CONTRACT OR GRANT NUMBER(s) (15) DAAA-25-75-C-0650
9. PERFORMING ORGANIZATION NAME AND ADDRESS Moore School of Electrical Engineering Department of Computer and Information Science University of Pennsylvania, Philadelphia, PA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 6.63.6.2A (17) 1G6 63622 AJ29 00 001
11. CONTROLLING OFFICE NAME AND ADDRESS US Army Electronics Command ATTN: DRSEL-TL-MS Fort Monmouth, NJ 07703		(16) 12. REPORT DATE (11) March 1978 13. NUMBER OF PAGES 369
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* (12) 380p.		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

The work covered by this report was sponsored by Frankford Arsenal, US Army, Philadelphia, PA. However, since the Frankford Arsenal mission relative to this work was transferred to the US Army Electronics Command (ECOM), Ft. Monmouth, NJ, this report is being issued as an ECOM report.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Automatic Test Systems (ATS)	Automatic Program Generation	Fault Isolation
Automatic Test Equipment (ATE)	Automatic Programming	Directed Graphs
Operational Performance Analysis Language (OPAL)	Precedence Relationships	Non Procedural
	Sequencing Strategies	
Non Procedural OPAL (NOPAL)	Sequence Ordering	

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The objective of the research is to design an automation aid for generating test programs that will (1) reduce the required user expertise, (2) reduce labor, (3) enhance efficiency, (4) improve user confidence, and (5) provide documentation for future utilization and maintenance.

The report presents a test description language, NOPAL, in which a user may describe diagnostic tests, and a software system which automatically generates test programs for an automatic test equipment based on the descriptions of tests

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

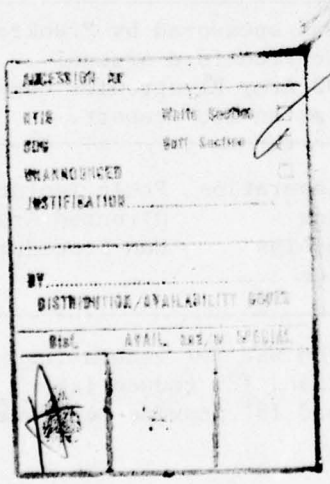
UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

410 039

20. ABSTRACT (continued)

The software system accepts as input the tests specified in NOPAL, performs syntax analysis, and constructs a directed graph. Based on the graph the system proceeds to check for consistency, completeness, and unambiguity in the supplied descriptions. The execution sequence of the tests is then optimized. Finally, an efficient program in a test equipment programming language, OPAL, is generated.



CONTENTS

ACKNOWLEDGMENT

The author is indebted to Professor Noah S. Prywes for his guidance and advice in the research and to the following for their programming and/or documentation help in the implementation of NOPAL Processor: Mr. Ron Berman for implementing intra-test-module analysis and sequencing (Summer and Fall 1976); Mr. Her-daw Che for his documenting NOPAL language and writing several subroutines dealing with dimension recognizer and NOPAL system monitor (Summer 1976).

CHAPTER 1
INTRODUCTION

1.1 Objective of Research

The ultimate objective of the research reported in this report is the automation of the programming task for automatic test systems.

This task is presently carried out by individuals who combine the skills of engineering and computer programming, and who generally employ an ad hoc approach. This task is laborious and frequently tiresome, which accounts for both the high costs involved and low confidence in the resulting products. The resulting low reliability of electronic and automotive equipment has had far-reaching economic and societal effects.

The first step towards automation of the test programming task has been the structuring of a respective methodology. For example, a typical analog electronic circuit test programming process can be summarized as follows:

(1) Obtain circuit diagram, manuals, and other design documents.

(2) Input the circuit to a circuit simulator, e.g., ECAP (Electronic Circuit Analysis Programs).

(3) Simulate failure mode of components of interest, and determine failure symptom.

(4) Evaluate potential tests that would have corresponding symptom in case of component failure.

(5) Prepare a flowchart for efficient conduct of test.

(6) Code test program.

(7) Debug the test program.

Note that a test engineer, who must also be a computer programmer, is responsible for carrying out this whole process manually. Thus an expert is absolutely needed to determine and specify tests. Moreover he must spend a considerable amount of time in coding and debugging the test program. The top and bottom parts of the ATS software component as depicted in Figure 1.1 are aimed at automating these various test steps, hence eliminating or reducing the above-mentioned problems.

1.2 Overview Of An Automatic Test System

In order to define the objectives more precisely it is necessary to digress very briefly and review the testing processes in the life of a product.

An Automatic Test System (ATS) consists of several elements such as the data base, circuit simulation, test generation, test language, and automatic test equipment [TO 74]. However, an ATS can be considered to be composed of two components: (1) hardware which consists of the

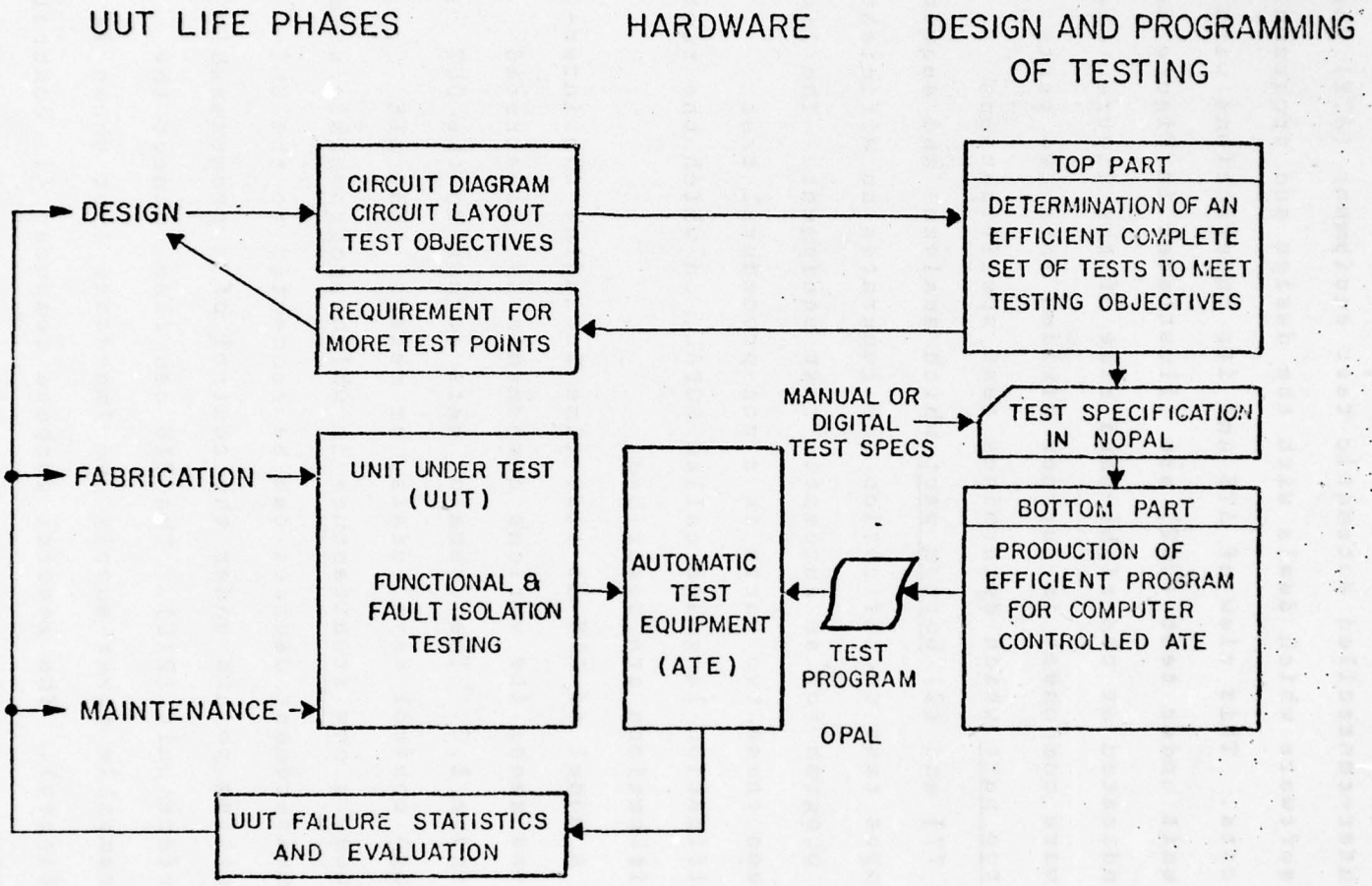


Figure 1.1 Design and Use of Testing in the Life Cycle of a Unit Under Test

computer-controlled automatic test equipment (ATE) and (2) software which deals with the design and programming of tests. This view of ATS and its interactions with the unit under test (UUT) are illustrated in Figure 1.1. As indicated at the right-hand side of the figure, the software component is further divided into two parts: (1) top part which determines test specifications [TIN 77] and (2) bottom part which analyzes and sequences an input test specification and generates an efficient test program for an automatic test equipment. The link between these two parts is a non-procedural test specification language called NOPAL, in which the test specifications are described.

A model of ATS at test-time including the interactions among its various components is illustrated in Figure 1.2. The operator gets access to the UUT through control keys, dials, or meters. The ATE includes a new architecture in which programmable stimuli and measurement devices can be connected to the UUT connecting points under the control of a programmable interface unit (PIU). The PIU can also connect the programmable power supply and impedance (not shown in the figure). The general purpose computer (1) controls the PIU switches, (2) sets those programmable units to

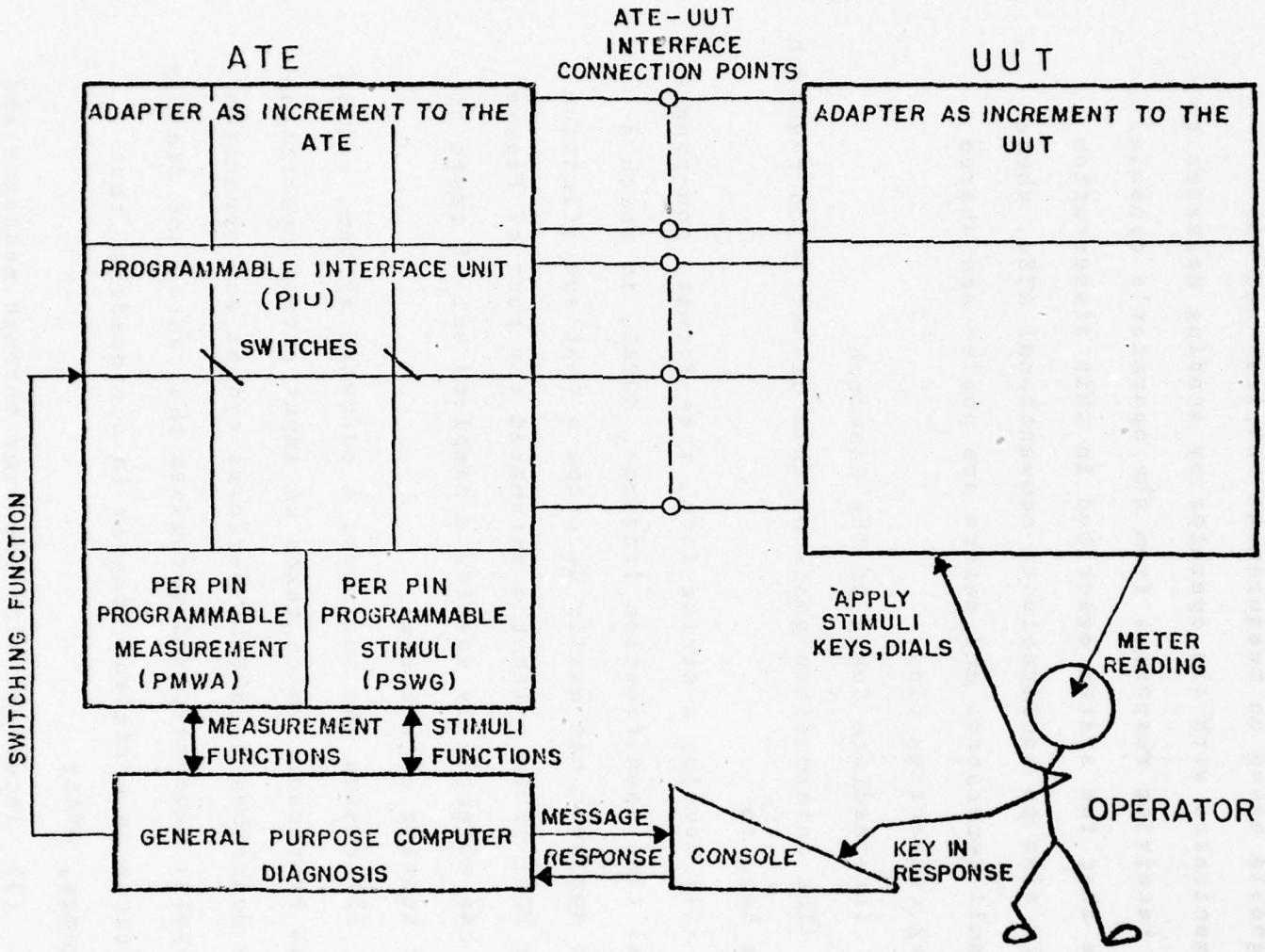


Figure 1.2 Illustration of the UUT-ATE Operation Test Cycle

perform the corresponding functions, (3) performs fault diagnosis based on measurement results, and (4) communicates with the operator by sending messages to and receiving responses from the operator's console. Note that the system described in this dissertation would also be applicable to conventional ATEs, where stimuli generators and meters are pooled and shared among connecting pins.

1.3 Intermediate Goals Of The Research

The intermediate goals of this dissertation research have been to:

(1) develop a string-form, free-format, non-procedural test specification language, NOPAL, in which a test engineer can easily describe a test specification of a UUT, or in which the automated top part of Figure 1.1 can effectively specify a complete set of tests to meet testing objectives;

(2) design and implement a software system, called NOPAL Processor, which takes as input a test specification described in NOPAL, performs syntax and semantics analysis, sequences and optimizes the tests, and finally generates an efficient program in a procedural test language, OPAL;

(3) Interface with the user through messages and documentation to facilitate specification preparation and program maintenance.

1.4 Overview of the NOPAL Language

NOPAL (Nonprocedural Operational Performance Analysis Language) is a specification language in which tests for various classes of devices can be described effectively. A complete test specification in NOPAL provides the information on the three major components of an ATS (as shown in Figure 1.2) at test time: (1) a set of test modules, (2) the UUT, and (3) the ATE.

The UUT specification describes (1) the UUT connecting points which can be connected to ATE through interface and (2) the potential UUT component failures.

The ATE section specifies (1) the ATE inter-connecting points which can be connected to matching UUT points and (2) all types of ATE functions: stimuli, measurement, failure, and evaluation which are referenced in the test modules.

A test module specification consists of (1) stimuli to be applied (2) measurements to be performed (3) logic to select the diagnoses, (4) operator message of each diagnosis, and (5) operator response of each diagnosis.

Instead of string form a NOPAL test specification can also be described in tabular form as presented in an earlier report [PRY 75, in which this author took part].

NOPAL is presented in Chapter 3 in more detail.

Several features of this language are extremely important in providing the ease and interactive features which are necessary for effective use.

First, it is non-procedural. The user can save much labor as compared to the use of procedural languages because he need not specify the execution order of events or the control logic. Nor does he need to specify storage declarations. These procedural actions will be deduced by the processor. All the statements are descriptive as opposed to imperative, and they can be entered in any order. The specification of each test in a multi test specification can be independently prepared. This independence of statements or test modules enables the user to concentrate on composing a single entity at a time. Also the test modules can be modified or added incrementally without considering the effect on other tests. Thus, virtually no computer programming knowledge is needed.

Second, NOPAL has the capability of self documentation. Various reports such as reformatted specification listing, cross references, error messages, and sequencing flowcharts are available to the user. This documentation will enhance the user-system interaction

or pinpoint erroneous spots in the user's specification.

Third, the NOPAL language and its processor are independent of both the units under test and the object automatic test equipments. Consequently NOPAL can be utilized to specify various classes of UUTs, such as hydraulic and mechanical, in addition to electronic devices. Of course, the simulation process (i.e. top part of Figure 1.1) must be re-designed for different classes of UUTs. Any changes in the ATE configuration have no effects on the NOPAL methodology. In this case, only the ATE functions (which are procedures defined in the object test language) should be properly modified if another set of ATE is to be used.

NOPAL is well-structured from top down. A test specification is functionally grouped into three sections: ATE, UUT, and test-modules. Each test module is then divided into three subsections: stimuli, measurement, and logic to select diagnoses. Finally a stimulus or measurement section is further broken into two distinct parts: (1) conjunction which involves UUT connection and ATE waveforms functions, and (2) assertions which perform pure computations. This simple structure enables the user to master the language easily. On the other hand, due to NOPAL's non-proceduralness and incrementality,

test modules can be composed independently from bottom up by a single user or a group of co-workers. Also, a user is relieved of the burden of explicitly specifying the execution sequence of test modules and the storage assignments, hence he can concentrate on what needs to be done rather than on how to do it. As a whole, the NOPAL methodology should enhance new understanding of programming process, and exemplify the top-down structured programming and team management techniques [MIL 71, BAK 72, STE 74].

The intuition and often unorganized thinking of the human being have been transformed into precise and systematic algorithms. For instance, based on the knowledge of how the test engineer sequences his test steps, several sequencing strategies such as data dependency and top-down fault isolation have been introduced. In addition some new algorithms have been developed. For example, a cycle enumeration and elimination algorithm is used in the phase of graph analysis, and several algorithms for invoking test module routines and inserting proper control logic are used in code-generation phase.

1.5 Importance and Contributions of the Research

The importance of this research is manifold. First, it affects real life -- it saves money. The research

will enhance an ATS in achieving the goal of reducing the enormous amount of maintenance and support costs on various complex systems, especially electronics systems. Currently, the cost ratio of computer software to hardware exceeds 2:1, and it has been forecast to approach 10:1 in ten years [e.g. BOE 74, PRY 74]. Thus any means of reducing the cost of software production in general is highly in demand. Particularly, the costs in maintenance and support of electronics systems are soaring, primarily as a result of systems complexity. For example, in 1973 the U.S. Department of Defense spent $15.7 billion on electronics, more than a third ($5.4 billion) of this amount were spent on maintenance and support. It has been estimated that 80% of maintenance goes to labor, and that about 87% of the repair time is required to locate and isolate defective components. Therefore the use of an ATS, which deals with diagnosis and fault isolation, can be estimated to affect almost 70% of the total maintenance cost.

This research also has its intellectual importance. Automatic testing is a complex and laborious task. Various aspects of the total ATS, such as UUT and its design/fabrication/maintenance life cycle, the functions of ATE, the role of an operator, and test determination need to be understood. The current exclusive role of human being in specifying and programming test procedures

should be replaced by an automation process. Thereby considerable amount of time and effort spent in programming and debugging can be saved. The expertise to play a dual role as engineer and programmer is no longer needed. Human errors will be much reduced. The understanding of a complex system and the transformation of the intuitive tasks into precise, concrete algorithms are important and significant in computer science in general.

The major contribution of this research is the development of a non-procedural test specification language NOPAL and the design and implementation of a software generation system (NOPAL Processor) which automatically produces reliable test programs from specifications in NOPAL.

The specification language is intended to be used easily by engineers to describe tests of UUTs manually, or by the automatic test determination process to specify tests systematically. Based on the test specification, the Processor will perform analysis on syntax and semantics, check for completeness and consistency, optimize execution sequence of test modules, and generate a complete test program. Therefore this system is expected to help reduce the amount of labor and necessary

expertise, program preparation and debugging times, human errors, etc., and ultimately help cut the enormous maintenance costs.

In summary, the NOPAL system is designed to eliminate the dual role of being an application programmer and the test engineer and provides a direct communication between the test engineer (or the automatic test determination process) and the ATE. It is a crucial step in the total automation of test determination and test program generation, hence has potential savings of manpower and costs needed for systems maintenance and support.

1.6 Organization of the Report

Chapter 2 summarizes the background and motivation of this research. Also included is a survey of related literature and research.

Chapter 3 is a user manual for the NOPAL language. It briefly presents the OPAL object language. The formal syntax of NOPAL and the overall structure of OPAL are also provided in an extended BNF notation.

Chapter 4 gives an overview of the NOPAL processor, the automatic test program generation system. It summarizes the three major phases of the Processor: (1) syntax analysis, (2) specification analysis, design, and

sequencing, and (3) code generation.

Chapter 5 describes the first phase of the NOPAL Processor. It presents in detail the syntax and statement analysis of the NOPAL specification, and the simulated associative memory where the specifications are to be stored. Included also are a meta-language EBNF/WSC, a meta-processor SAPG, and a formal syntax of NOPAL in EBNF/WSC.

Chapter 6 covers the second phase of the NOPAL Processor - specification analysis, design and sequencing. It includes the theoretical background, methodology and the algorithms of the intra and inter test-module analysis and sequencing.

Chapter 7 presents the last phase of the Processor, code generation. At this phase the analyzed and sequenced test specification is encoded into a complete test program in an object language, in particular, OPAL. Algorithms and data structures needed in this code-generation phase are outlined.

Chapter 8 summarizes the conclusions that can be drawn from this research, and suggests future research areas.

At the end, an appendix is provided with a list of data structures used in the NOPAL system, and a sample NOPAL test specification together with its corresponding output which has been produced by the Processor.

CHAPTER 2

SURVEY OF RELATED LITERATURE AND RESEARCH

2.1 Introduction

As already indicated, this research has been primarily motivated by two considerations. First was the concern over the soaring costs of software development. This consideration is shared by research on automatic programming. For example, it is believed that the entire software development process could be automated effectively. Section 2.2 summarizes the research on automation of software development in various applications.

The second consideration concerned particularly the widespread need for test programs utilizing a rapidly advancing technology of automatic test systems (ATS). Section 2.3 surveys automatic testing generally and Section 2.4 presents some typical programming languages for automatic test equipments.

As will be shown, while there has been great progress in facilitating software development through automation, the application of such techniques in automatic testing is lagging considerably behind their use in other areas. The progress in automatic testing has been impressive in new hardware techniques, but the software state-of-the-art in automatic testing has not

experienced similar progress. One of the objectives of the research reported here has been to correct this unbalance.

2.2 Research on Automation of Software Development Process

Recently there has been a concerted effort to automate software development process systematically. Typically, this process can be broken into several phases as outlined below [TEI 71, COU 73, PRY 74].

(1) Determination of problem requirements

(2) Functional specifications of the system

(3) Physical design of the system, including system module decomposition and input/output specifications

(4) Program design

(5) Program implementation: coding, debugging, and documentation

(6) System integration and installation

(7) Operation, maintenance, and modification

Although some progress has been made on the automation of the first two phases, the interest here is in research on the automation of the remaining phases. This is summarized below.

Early theoretical work to develop design techniques includes an information algebra [COD 62], a machine-independent problem-defining language for systems analysts. Langefors [LAN 63] also introduced some theoretical approaches to information systems.

Although the evolution of computer software techniques has always lagged that of computer hardware [COU 73], many automation facilities such as assemblers, compilers and operating systems have been developed to help reduce human laborious effort and errors. In addition there have been important developments to aid system specification and documentation. Several manual techniques such as SOP [IBM 61] and ADS [LYN 69] have been designed for this purpose. Later some of these languages were supported by automation. For example, ADS processors [THA 71] were developed but were not used extensively. IBM's TAG system (Time Automated Grid) was developed in 1962 and automated later [IBM 71]. Another language useful to systems analysts in designing information models was systematics [GRI 66].

Developed at the University of Michigan were a problem statement language (PSL) for describing system requirements and a problem statement analyzer (PSA) to analyze and produce program specifications [TEI 71, TEI 74, TEI 76]. Nunamaker extended PSL/PSA into

SODA (Systems Optimization and Design Algorithm), whose objective was to generate a complete system design from a statement of the processing requirements [NUN 71]. The use of computer-aided analysis for the design and development of an integrated financial management system using SODA and ADS was exemplified in [NUN 76].

Research on automatic program generation has been pursued for years at the University of Pennsylvania. In 1973, a DDL/DML system was developed to automate the generation of data conversion programs [RAM 73]. Since then the research effort has been directed to the development of two automatic program generation systems: NOPAL and MODEL.

The NOPAL system, which is the topic of this dissertation, was introduced to achieve the goal of total automation of test determination and test program generation for computer-controlled automatic test equipments [PRY 75]. The system consists of two independent parts: (1) top part -- test determination of electronic circuits and (2) bottom part -- automatic test program produccion. The top part determines complete test specifications expressed in the NOPAL language [TIN 77]. The bottom part, described in this dissertation accepts as input the test specifications written in the NOPAL

language and produces as output an efficient test program in a procedural test programming language OPAL. The bottom part can also accept test specifications manually prepared by a user.

A MODEL system was developed in 1974-5. It generates PL/1 programs for business applications of transaction processing [RIN 76]. A revised version, called MODEL II is an operational system with a simplified user-oriented language [PRY 77a]. A new version of MODEL is also under development [GAN 75, SHA 76, PRY 77b]. It includes interactiveness, tolerance of incompleteness or ambiguities, and modelling and forecasting.

NOPAL and MODEL share several common techniques such as using Syntax Analysis Program Generator to produce a syntax parser, encoding statements in a simulated associative memory to facilitate later retrieval, and constructing directed graph models for sequencing and analysis purpose. But they differ in many aspects. MODEL is designed for general data processing and emphasizes computational capabilities, while NOPAL is for engineers and hence focuses more on providing engineering facilities. NOPAL incorporates engineering knowledge base and fits a particular class of user, engineer. MODEL provides more complex iteration facility and has more data types.

Extensive research on automatic program generation has been conducted at MIT. Research on automation of various phases from problem definition to program compilation and execution has been attempted simultaneously. These efforts have been integrated in an automatic programming system prototype, PROTOSYSTEM I, which consists of top and bottom parts. The top part deals with problem definition and system analysis and design. It represents (1) an automated consultant system for operations management [HAX 73, MAR 76], (2) the research on questionnaire approaches [MAL 75], and (3) the development of the OWL language for user communication [MAR 74]. The bottom part addresses the problems of implementation and optimization of a program, given an abstract specification of what to be done [RUT 76]. Thus it accepts a specification from the top part, performs program design and optimization, and finally generates PL/1 code and the needed job control statements.

The MIT research has tried to specialize in assisting the operations management consultant in solving procurement problems. In contrast, NOPAL system provides engineers with an automation aid in producing test programs.

2.3 Literature on Automatic Testing and Automation Test Equipments

The advent of solid-state technology has made the electronic components and systems more complex. Consequently, the problems associated with testing and the testability of electronic devices and systems continue to grow in complexity. Sophisticated testing is absolutely necessary in order to keep the equipment operational. The soaring maintenance and support costs on these systems as illustrated in Chapter I have also prompted the development of automatic testing -- isolating faults quickly and with minimum human intervention.

While automatic testing can be used in many other fields such as mechanical, hydraulic and optical, the technique has been widely employed in testing electronic equipment. Therefore, the subsequent discussions focus primarily on the automatic testing of electronic devices.

Automatic testing, when applied to electronic circuits, is designed to evaluate three types of circuits: a) digital, (2) analog, and (3) hybrid. There are three basic testing methods for digital circuits [McA 71, ELE 74]:

(1) Functional testing, the application of binary signals to determine Boolean operation.

(2) Static or parametric testing, to check static characteristics such as DC parameters.

(3) Dynamic testing, to check time-dependent parameters such as rise and fall times, and propagation delay.

When applied to analog circuits, functional testing means to check if the systems perform well according to design. While there has been considerable research and development on simulation of digital circuits, there has been little work done on generating tests for analog circuits [GRE 73].

An automatic test system consists of several main components [TO 74]:

(1) Data base of the descriptions of the unit under test, e.g., electronic circuit

(2) Means of simulation

(3) Test generation techniques

(4) Command language and program library

(5) Automatic test equipment (ATE)

(6) Fault isolation

(7) Data management

To achieve maximum effectiveness of an ATS, a concerted effort must be directed to all of these components.

Automatic test equipment ranges from simple fixed program comparator or pattern generator to the variable stored program, computer-controlled test system. Some ATE units are briefly described below.

Automatic test systems have been largely used by military and semiconductor manufacturers [LUS 73], and are beginning to be used in almost every field, including industrial control, manufacturing, and computer networks.

A good example is the electronic switching systems (ESS) used in the Bell Telephone System [BEL 70, BEL 73]. To achieve an expected down time of two hours in forty years, Bell Telephone has counted on such automatic testing methods as using central control offices to initiate fault analysis automatically.

The industrial spectrum of automatic testing as outlined in the following paragraph was summarized from [ELE 74 and SEM 74].

At Honeywell automatic testing is used to develop some new products. Westinghouse has initiated systems in transportation, remote terminal, and contactless

testing. Adar Associates has been concentrating on semiconductor memory and micro processor testing. Both Teradyne and Fairchild Systems Technology produce computer-controlled IC testers and modularized test systems. Finally System 390 by Instrumentation, S-3260 by Tektronix, and 9500 Series by Hewlett-Packard are the typical fully automatic test systems.

A remarkable automatic test station, SCATE MARK VI [GEN 73] was introduced by General Dynamics. It is computer-controlled, modularized, and capable of testing digital, analog, or RF electronics at frequencies up to 1.3 GHz. The system includes a programmable interface unit, a sampling measurement system, and an arbitrary stimulus function generator. Thereby conventional adaptors, measurement instruments, and stimulus generators are eliminated. Moreover, it uses a high level test programming language (ATLAS). The ATE architecture depicted in Figure 1.2 has been influenced by the recent advances in the ATE units such as SCATE MARK VI.

Many aspects of automatic testing such as software and languages, simulation, interfacing fault isolation, failure prediction, and self-repairing circuits are gaining more attention. For example, Institute of

Electrical and Electronics Engineering (IEEE) has held many forums (e.g. INTERCON and WESCON) and yearly symposia (e.g. Automatic Support Systems Symposium and Semiconductor Test Symposium [SEM 74]). A formal publication on automatic test equipments was also sponsored by IEEE [LIG 74].

2.4 Survey of ATE Programming Languages

Programming languages are used in stored program, computer-controlled automatic test equipments to write test procedures. Some typical high level, procedural test languages are summarized in the following paragraphs.

Probably the most popular test language is ATLAS (Abbreviated Test Language for Avionic Systems) a language originally developed for airline avionic equipment testing [ARI 76]. Many modified versions of ATLAS also exists. For example, a version called EQUATE-ATLAS was modified by RCA for the U.S. Army EQUATE system [RCA 73].

CTL (Compass Test Language) is another test language developed for the U.S. Army [WAR 74]. It was designed to include some features of modern high level programming languages such as looping constructs and decision tables. Also it could be used to program tests for UUTs other than avionic or electronic equipments.

GOAL (Ground Operations Aerospace Language) was developed at NASA JFK Space Center [NAS 73]. It is a test engineer oriented language, in which an engineer can program test procedures for space shuttle pre-flight checkout, ground preflight operations, etc.

Most recently, OPAL(Operational Performance Analysis Language) was developed for the U.S. Army (Frankford Arsenal) [FRA 76a]. It is a language in which one can program tests for various classes of devices. Supposedly its design is based upon the concepts derived from ATLAS, CTL, GOAL, and other general purpose languages such as ALGOL and PL/1. OPAL is the objective language of the automatic test program generation system reported in this dissertation and is covered in Chapter 3 in more detail.

All the above-mentioned test languages are high level, procedural. The user has to sequence his test steps and takes care of storage assignments, etc. In essence, they are like general purpose, high level programming languages except that they also provide test-oriented capabilities. Therefore the higher level or non-procedural NOPAL language was introduced. The user can describe rather than program his tests in this language, hence lower level programming details are eliminated.

CHAPTER 3

THE NOPAL AND OPAL LANGUAGES

3.1 Introduction

The objective of this chapter is to present the source and target languages of the automatic test program generation system. Sections 3.2 and 3.3 describe the source language NOPAL (Non-procedural OPAL) and Section 3.4 presents the target language OPAL (Operational Performance Analysis Language).

NOPAL is a very high-level, non-procedural language in which a set of tests and diagnoses for a unit under test (UUT) can be easily described. It is non-procedural in the sense that its statements are descriptive and can be entered in any order, that it does not provide facilities for stating commands or sequencing and control logic, and that additional statements can be incrementally included. This language was developed to facilitate the process of automated design and programming of testing.

OPAL, in which the object program outputted from this system is written, is a high level, procedural test language for programming automatic test equipments (ATE). Its definition, syntax, and semantics have been fully documented [FRA 76]. This documentation should be

referred to for more details; Section 3.4 only highlights this language.

One major disadvantage for a test engineer to use OPAL directly is that he has to "program" the tests rather than "describe" them. He must also be a computer programmer, sequence his test steps, plan storage assignments, and so forth. Therefore NOPAL was introduced to overcome these drawbacks by substituting an automatic process for human programmers. NOPAL is a specification scheme in which the test engineer can describe what to be done disregarding how to do it. He should be able to master this methodology fast and easily. On the other hand, an automatic test determination process can effectively generate test specifications described in NOPAL. The burden of lower-level details is left to the NOPAL processor.

3.2 The NOPAL Language

This section describes the syntax and semantics of the NOPAL (Non-procedural OPAL) language. NOPAL is a description language in which test specifications can be described effectively.

A collection of NOPAL statements describing a functional module is referred to as a specification. A complete description, in NOPAL, of the tests desired

for a given UUT and under a given ATE is referred to as a NOPAL specification or test specification. NOPAL statements are delimited by statement end markers (;). They can appear in any order due to the nature of non-proceduralness. Yet for organization purposes, each test specification can be divided into the following three major sections immediately after a specification header statement which identifies the whole specification:

(1) test-modules specification which describe the set of desired tests including the stimuli to be applied, measurements to be made, logic to be used for selecting diagnoses, and definitions of the diagnoses and messages;

(2) UUT specification whose statements identify the potential component failures and connection points of the UUT;

(3) ATE specification which defines the ATE functions (such as stimuli, measurements, and special-purpose computational) and the ATE inter-connecting points.

The last two sections are described in Sections 3.2.4 and 3.2.5 respectively. The test-modules specification is discussed separately in Section 3.3 due to its complexity.

Section 3.2.1 introduces a new syntax notation, which is extended from conventional Backus Normal Form, and the complete formal syntax of NOPAL in this meta-language.

To make it easier to explain the NOPAL language a sample test specification is presented in Section 3.2.2.

Section 3.2.3 summarizes the fundamental lexical constructs which are used to form higher level NOPAL statements.

Note that in the NOPAL language, if a keyword is composed of more than five characters, only the first four characters are significant (and such keywords are listed in Table 5.5 of Chapter 5).

3.2.1 Extended BNF Syntax Notation

In the subsequent descriptions of the syntax of NOPAL statements, a meta-language EBNF (Extended Backus Normal Form) is used. EBNF and its processor were developed at the University of Pennsylvania by the DDL project [RAM 73 and FRE 72]. They are further discussed in Section 5.2 of Chapter 5. As in conventional BNF, names enclosed by angle brackets < >, called non-terminals, represent syntactic classes for which specific items are eventually to be substituted, and non-bracketed

names represent terminal symbols. The symbol ::= means "is defined as" or "produces". The vertical bar | denotes "or", and is used for alternatives. The two extensions to the conventional BNF are optionality and repetition. Units enclosed in square brackets [ ] indicate that they are optional, i.e., they may appear zero or one time. If an asterisk also follows the right square bracket, i.e. [ ]*, it means that the enclosed item may be repeated zero or more times.

The complete formal syntax of NOPAL in EBNF is presented in Figure 3.1. To enhance readability, EBNF statements (productions) are indented and assigned level numbers to indicate the depth within the syntax-tree structure. Numbers to the right of the EBNF specification are the corresponding EBNF statement (or production) numbers, which will be referred to in later discussions.

3.2.2 A Sample NOPAL Specification - MINIRADIOSET

In order to facilitate the discussion of the various aspects of the NOPAL language and demonstrate its usage, a sample test specification written in NOPAL is presented in Figure 3.2. This specification (referred to as MINIRADIOSET) along with a variety of reports produced by the NOPAL processor can also be found

```
1 <STRING_CONST>  ::=  <CHAR_STRING>  | <BIT_STRING>
  2 <CHAR_STRING>  ::=  '[<FULL_CHAR>]*'
    3 <FULL_CHAR>  ::=  <LETTER> | <DIGIT> | <SPECIAL_CHAR>
      4 <LETTER>  ::=  A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|_
      4 <DIGIT>  ::=  0|1|2|3|4|5|6|7|8|9
      4 <SPECIAL_CHAR>  ::=  .|<|(|+|||&|!|$|*|)|;|¬|-|/|,|%|_|>|?|:|#|@|'|=|"|BLANK
  2 <BIT_STRING>  ::=  '<BIT> [<BIT>]*'B
    3 <BIT>  ::=  0|1

1 <NUMBER>  ::=  [<SIGN>] <UNSIGNED_NUMBER>
  2 <SIGN>  ::=  +|-
  2 <UNSIGNED_NUMBER>  ::=  <DECIMAL_NUMBER> [<EXPONENT>]
    3 <DECIMAL_NUMBER>  ::=  <UNSIGNED_INTEGER> [<DECIMAL_FRACTION>]
                          | <DECIMAL_FRACTION>
      4 <UNSIGNED_INTEGER>  ::=  <DIGIT> [<DIGIT>]*
      4 <DECIMAL_FRACTION>  ::=  .<UNSIGNED_INTEGER>
    3 <EXPONENT>  ::=  E[<SIGN>] <DIGIT> [<DIGIT>]

1 <INTEGER>  ::=  [<SIGN>] <UNSIGNED_INTEGER>

1 <VARIABLE>  ::=  <IDENTIFIER> [(<SUBSCRIPT_LIST>)]
  2 <IDENTIFIER>  ::=  <LETTER> [<TAIL>]*
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL

```
   3 <TAIL>  ::=  <LETTER> | <DIGIT>                                          19
  2 <SUBSCRIPT_LIST>  ::=  <ARITH_EXPR> [,<ARITH_EXPR>]*                      20

1 <ARITH_EXPR>  ::=  [<SIGN>] <TERM> [<ADD_OP> <TERM>]*                      21
  2 <TERM>  ::=  <FACTOR> [<MULT_OP> <FACTOR>]*                              22
    3 <FACTOR>  ::=  <PRIMARY> [** <PRIMARY>]*                               23
      4 <PRIMARY>  ::=  <UNSIGNED_NUMBER> | <VARIABLE> | <FUNCTION_CALL>     24
                      | (<ARITH_EXPR>)
        5 <FUNCTION_CALL>  ::=  <FUNCTION_ID> [(<ARGUMENT>  ,<ARGUMENT> *)]  25
          6 <FUNCTION_ID>  ::=  <IDENTIFIER>                                 26
          6 <ARGUMENT>  ::=  <ARITH_EXPR> | <STRING_CONST>                   27
    3 <MULT_OP>  ::=  * | /                                                  28
  2 <ADD_OP>  ::=  + | -                                                     29

1 <COMMENT>  ::=  /* [<FULL_CHAR>]* */                                       30

1 <IF_CLAUSE>  ::=  IF <BOOLEAN_TERM> THEN                                   31
  2 <BOOLEAN_TERM>  ::=  <BOOLEAN_FACTOR> [V <BOOLEAN_FACTOR>]*              32
    3 <BOOLEAN_FACTOR>  ::=  <BOOLEAN_PRIMARY> [& <BOOLEAN_PRIMARY>]*        33
      4 <BOOLEAN_PRIMARY>  ::=  <RELATIONAL_EXPR> | ¬ (<BOOLEAN_TERM>)       34
        5 <RELATIONAL_EXPR>  ::=  <ARITH_EXPR> <RELATION> <ARITH_EXPR>       35
          6 <RELATION>  ::=  = | > | < | >= | <= | ¬= | ¬> | ¬<               36
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
1 <CONN_DIM_EX>  ::=  <CONNECTOR> [<DIMENSION>]                                    37
  2 <CONNECTOR>  ::=  <CONNECTOR_ID> | < <CONNECTOR_ID> [,<CONNECTOR_ID>]* >      38
    3 <CONNECTOR_ID>  ::=  <OUT_POINT_ID>                                          39
  2 <DIMENSION>  ::=  [<PREFIX>] <BASIC_UNIT> [/SEC/SEC] | <TIME_DIMENSION>       40
    3 <PREFIX> ::= <|G|M|C|U|N|P|MEG                                               41
    3 <BASIC_UNIT> ::= G|M|CD|CM|DB|FD|FT|GM|IN|HP|HZ|LB|LI|LM|PC|AMP|BAR|DEG|ERG|GAL   42
                       |LUX|OHM|PPM|PSI|RAD|RPH|RPM|RPS|CU_M|DEGC|DEGF|DYNE|INHG|LINE
                       |MMHG|NEWT|SQ_MI|STER|VOLT|WATT|CU_FT|FT_LB|HENRY|JOULE|IND_IP
                       |POUNDAL|BRAKE_HP|%
    3 <TIME_DIMENSION>  ::=  [<PREFIX>] <TIME_UNIT>                                43
      4 <TIME_UNIT> ::= HR|MIN|SEC                                                 44

1 <VAL_DIM_EX>  ::=  <ARITH_EXPR> [<DIMENSION>]                                    45

1 <FUNC_DIM_EX>  ::=  <FUNC_TERM> [<ADD_OP> <FUNC_TERM>]*                          46
  2 <FUNC_TERM>  ::=  <FUNC_FACTOR> [<MULT_OP> <FUNC_FACTOR>]*                     47
    3 <FUNC_FACTOR>  ::=  [<FUNC_MODIFIER> <MULT_OP>] <FUNC_PRIMARY>               48
      4 <FUNC_MODIFIER>  ::=  <UNSIGNED_NUMBER>                                    49
      4 <FUNC_PRIMARY>  ::=  <FUNCTION_ID> [(<FUNC_ARG> [,<FUNC_ARG>]* )]          50
                           | (<FUNC_DIM_EX>)
        5 <FUNC_ARG>  ::= [<RELATION>] <VAL_DIM_EX> | [=] <RANGE> | <STRING_CONST> | *   51
          6 <RANGE>  ::=  <VAL_DIM_EX>  +- <VAL_DIM_EX> | <VAL_DIM_EX> +- <ARITH_EXPR> [%]   52
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
1 <NOPAL_SPECIFICATION>  ::= [NOPAL] SPECIFICATION [<SPEC_NAME>] |                    53
                             [<NOPAL_STMTS>]*
                             END [<SPEC_NAME>] |
  2 <SPEC_NAME>  ::= <LABEL>                                                         54
    3 <LABEL>  ::= <IDENTIFIER> | <UNSIGNED_INTEGER>                                 55
  2 <NOPAL_STMTS>  ::= <TEST_MODULE_SPEC> | <UUT_SPEC> | <ATE_SPEC>                  56
    3 <TEST_MODULE_SPEC>  ::= <TEST_STEP>                                           57
                            | <DIAGNOSIS_DEFINITION> [<DIAGNOSIS_DEFINITION>]*
                            | <MESSAGE_DEFINITION> [<MESSAGE_DEFINITION>]*
      4 <TEST_STEP>  ::= TEST [<TEST_LABEL>] |                                      58
                       | STIMULI <STIM_ID> [:<BACK_REFERENCE>[<DECLARATION>]*] |
                       | MEASUREMENT <MEAS_ID> [:<BACK_REFERENCE>[<DECLARATION>]*]|
                       | LOGIC <LOGIC_ID> [:] <LOGIC_DIAG_LIST> |
                       | <WAVEFORMS>
      5 <TEST_LABEL>  ::= <LABEL>                                                   59
      5 <STIM_ID>  ::=  <LABEL>  [(<TEST_LABEL>)]                                   60
      5 <MEAS_ID>  ::= <STIM_ID>                                                    61
      5 <LOGIC_ID>  ::= <STIM_ID>                                                   62
      5 <LOGIC_DIAG_LIST>  ::= <LOGOP_DIAGLBL> [,<LOGOP_DIAGLBL>]*                  63
        6 <LOGOP_DIAGLBL>  ::= <LOGICAL_OPERATOR> <DIAG_LABEL>                      64
          7 <LOGICAL_OPERATOR>  ::= <LOGICAL_CONNECTIVE> [<AFTER>] | <AFTER>        65
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
      8 <LOGICAL_CONNECTIVE>  ::=  V | & | V¬ | &¬ | *                    66
      8 <AFTER>  ::=  A | A¬                                              67
5 <WAVEFORMS>  ::=  <CONJUNCTION> | <ASSERTION> [<ASSERTION>]*             68
  6 <CONJUNCTION>  ::=  CONJUNCTION <WAVEFORM_ID> : <CONJUNCTION_BODY>     69
                              [<DECLARATION>]* ;
    7 <WAVEFORM_ID>  ::=  [<LABEL>] [(<STIM_MEAS_LABEL>)]                 70
      8 <STIM_MEAS_LABEL>  ::=  <LABEL>                                   71
    7 <CONJUNCTION_BODY>  ::=  <TRIPLET_CONJUNCT> | <BACK_REFERENCE>      72
      8 <TRIPLET_CONJUNCT>  ::=  <SIMPLE_CONJUNCTION> | <IF_CONJUNCTION>  73
        9 <SIMPLE_CONJUNCTION>  ::=  <TRIPLET> [& <TRIPLET>]*             74
          10 <TRIPLET>  ::=  (<CONN_DIM_EX> <RELATION> <FUNC_DIM_EX>)     75
                           | <CONN_DIM_EX> <RELATION> <FUNC_DIM_EX>
        9 <IF_CONJUNCTION>  ::=  <IF_CLAUSE><SIMPLE_CONJUNCTION>[ELSE<TRIPLET_CONJUNCT>]  76
      8 <BACK_REFERENCE>  ::=  [SAME] AS <STIM_MEAS_LABEL>                77
                              [EXCEPT <SIMPLE_CONJUNCTION>]
    7 <DECLARATION>  ::=  <VARIABLE_TYPE> [:] <VARIABLE_LIST>             78
      8 <VARIABLE_TYPE>  ::=  SOURCE | TARGET                             79
      8 <VARIABLE_LIST>  ::=  (<VARIABLE> [,<VARIABLE>]*)                 80
                           | <VARIABLE> [,<VARIABLE>]*
  6 <ASSERTION>  ::=  ASSERTION <WAVEFORM_ID> : <ASSERTION_BODY>          81
                              [<DECLARATION>]* ;
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
 7 <ASSERTION_BODY>  ::=  <SIMPLE_ASSERTION> | <IF_ASSERTION>                          82
   8 <SIMPLE_ASSERTION>  ::=  <RELATIONAL_EXPR>                                       83
                            | <ARITH_EXPR> = <ARITH_EXPR> +- <ARITH_EXPR> [%]
   8 <IF_ASSERTION>  ::= <IF_CLAUSE> <SIMPLE_ASSERTION> [ELSE <ASSERTION_BODY>]     84
4 <DIAGNOSIS_DEFINITION>  ::=  DIAGNOSIS <DIAG_LABEL> [:] <DIAG_BODY> |             85
  5 <DIAG_LABEL>  ::=  <LABEL>                                                      86
  5 <DIAG_BODY>  ::=  <POSITIONAL_DIAG> | <KEYWORDED_DIAG>                          87
    6 <POSITIONAL_DIAG>  ::=  [<OPERATOR_MESSAGE>] [,<OPERATOR_RESPONSE>]           88
      7 <OPERATOR_MESSAGE>  ::=  ([<AFFECTED_COMPONENTS>] [,[<OTHER_PARAMETERS>]    89
                                 [,[<TYPE>] [,<TIMING>]]] )
                               | <AFFECTED_COMPONENTS>
      8 <AFFECTED_COMPONENTS>  ::=  <COMPONENT_CONJUNCT> [& <COMPONENT_CONJUNCT>]*  90
                                  | <COMPONENT_DISJUNCT> [V <COMPONENT_DISJUNCT>]*
        9 <COMPONENT_CONJUNCT>  ::=  <COMP_FAIL_SEQN> | <COMPONENT>                 91
                                   | <FAILURE_FUNCTION> ( <COMPONENT> [& <COMPONENT>]* )
         10 <COMPONENT>  ::=  <IDENTIFIER>                                          92
        9 <COMPONENT_DISJUNCT>  ::=  <COMP_FAIL_SEQN> | <COMPONENT>                 93
                                   | <FAILURE_FUNCTION> ( <COMPONENT> [V <COMPONENT>]* )
      8 <OTHER_PARAMETERS>  ::=  ( <MSG_ARGUMENT> [,<MSG_ARGUMENT>]* )             94
                               | <MSG_ARGUMENT>
      9 <MSG_ARGUMENT>  ::=  <STRING_CONST>  |  <VARIABLE> |  <NUMBER>             95
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
      8 <TYPE>  ::=  <MESSAGE_LABEL>                                          96
      8 <TIMING>  ::=  <NUMBER> [<TIME_DIMENSION>]                            97
    7 <OPERATOR_RESPONSE>  ::=  Y/N | (<OP_VAR_LIST>) [,] [Y/N]               98
                              | <OP_VAR_LIST> [Y/N]
      8 <OP_VAR_LIST>  ::=  <VARIABLE> [,<VARIABLE>]*                         99
  6 <KEYWORDED_DIAG>  ::=  [OPERATOR MESSAGE:] <DIAG_KEYWORD> [,<DIAG_KEYWORD>]* 100
    7 <DIAG_KEYWORD>  ::=  [AFFECTED] COMPONENT = <AFFECTED_COMPONENTS>       101
                         | [OTHER] PARAMETER = <OTHER_PARAMETERS>
                         | TYPE = <TYPE>
                         | TIME = <TIMING>
                         | RESPONSE = <OPERATOR_RESPONSE>
  4 <MESSAGE_DEFINITION>  ::=  MESSAGE <MESSAGE_LABEL> [:]                    102
                         [ALIAS = <SYNONYM>,] [TEXT=] <MESSAGE_TEXT>:
    5 <MESSAGE_LABEL>  ::=  <LABEL>                                           103
    5 <SYNONYM>  ::=  <IDENTIFIER>                                            104
    5 <MESSAGE_TEXT>  ::=  <TEXT_ELEM> [[,]<TEXT_ELEM>]*                      105
      6 <TEXT_ELEM>  ::=  <CHAR_STRING>                                       106

3 <UUT_SPEC>  ::=  <UUT_COMPONENT_FAILURE> [<UUT_COMPONENT_FAILURE>]*         107
                | <UUT_CONNECTION_POINT> [<UUT_CONNECTION_POINT>]*
  4 <UUT_COMPONENT_FAILURE>  ::=  COMP_FAIL[<COMP_FAIL_SEQ#>] [:]<COMPONENT>  108
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
                                        [,<COMP_FAIL_KEYWD>]* ;
  5 <COMP_FAIL_SEQ#>  ::=  <ENTRY_SEQ#>                                    109
    6 <ENTRY_SEQ#>  ::=  <UNSIGNED_INTEGER>                                110
  5 <COMP_FAIL_KEYWD>  ::=  ALIAS = <SYNONYM>                              111
                          | FAILURE [FUNCTION] = <FAILURE_FUNCTION>
                          | PARAMETER= <PARM_LIST> | INDEX= <FAILURE_INDEX>
                          | PROTECTION= <PROTECTION> | <COMMENTS>
    6 <FAILURE_FUNCTION>  ::=  <FUNCTION_ID>                               112
    6 <PARM_LIST>  ::=  (<PARM_NAME> [, <PARM_NAME>]* ) | <PARM_NAME>      113
      7 <PARM_NAME>  ::=  <IDENTIFIER>                                     114
    6 <FAILURE_INDEX>  ::=  <INTEGER>                                      115
    6 <PROTECTION>  ::=  (<COMP_FAIL_ID> [<COMP_FAIL_ID>]*) | <COMP_FAIL_ID>  116
    6 <COMP_FAIL_ID>  ::=  <COMP_FAIL_SEQ#> | <COMPONENT> |                117
                           <FAILURE_FUNCTION>(<COMPONENT>)
    6 <COMMENTS>  ::=  [COMMENT=] <CHAR_STRING>                            118
4 <UUT_CONNECTION_POINT>  ::=  UUT_POINT [<ENTRY_SEQ#>] [:] <UUT_POINT_ID>  119
                                        [,<UUT_POINT_KEYWD>]* ;
  5 <UUT_POINT_ID>  ::=  <IDENTIFIER>                                      120
  5 <UUT_POINT_KEYWD>  ::=  ALIAS = <SYNONYM>                              121
                          | CONNECTOR = <UUT_CONNECTOR>
                          | LIMIT = <PROTECTIVE_LIMITS> | <COMMENTS>
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
6 <UUT_CONNECTOR>  ::=  (<CONN_TYPE> [, <CONN_POINT>]) | <CONN_TYPE>          122
  7 <CONN_TYPE>  ::=  <IDENTIFIER>                                            123
  7 <CONN_POINT>  ::=  <IDENTIFIER>                                           124
6 <PROTECTIVE_LIMITS>  ::=  ([<DIMENSION>] [, <MAX_LIMIT>] [,[<MIN_LIMIT>]   125
                                                [,<REFERENCE_POINT>]])  |
                          | <DIMENSION>
  7 <MAX_LIMIT>  ::=  <NUMBER>                                                126
  7 <MIN_LIMIT>  ::=  <NUMBER>                                                127
  7 <REFERENCE_POINT>  ::=  <UUT_POINT_ID>                                    128

3 <ATE_SPEC>  ::=  <ATE_FUNCTION> [<ATE_FUNCTION>]*                           129
                | <ATE_CONNECTION_POINT> [<ATE_CONNECTION_POINT>]*
  4 <ATE_FUNCTION>  ::=  FUNCTION [<ENTRY_SEQ#>] [:] <FUNCTION_ID>            130
                                        [,<FUNCTION_KEYWD>]*  |
    5 <FUNCTION_KEYWD>  ::=  ALIAS = <SYNONYM>                                131
                           | [FUNCTION] TYPE= <FUNCTION_TYPE>
                           | #PINS= <UNSIGNED_INTEGER>
                           | PARAMETER=<PARM> [,PARAMETER=<PARM>]*
                           | VALUE [RETURNED] = <VALUES_RETURNED>
                           | COOPERATION = <COOP_FUNCTIONS> | <COMMENTS>
    6 <FUNCTION_TYPE>  ::=  S | M | F | E | C                                 132
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
6 <PARM>  ::=  (<PARM_NAME> [,[<PARM_TYPE>] [,[LIMIT=]<PARM_LIMITS>]] )      133
            | <PARM_NAME>
  7 <PARM_TYPE>  ::=  S | T                                                  134
  7 <PARM_LIMITS>  ::=  ([<DIMENSION>] [,[<MAX_LIMIT>] [,<MIN_LIMIT>]] )    135
                      | <DIMENSION>
6 <VALUES_RETURNED>  ::=  <CHAR_STRING>                                      136
6 <COOP_FUNCTIONS>  ::=  (<FUNCTION_ID> [, <FUNCTION_ID>]*) | <FUNCTION_ID>  137
4 <ATE_CONNECTION_POINT>  ::=  ATE_POINT [<ENTRY_SEQ#>][:] <ATE_POINT_ID>    138
                                  [,<ATE_POINT_KEYWD>]* (
  5 <ATE_POINT_ID>  ::=  <IDENTIFIER>                                        139
  5 <ATE_POINT_KEYWD>  ::=  ALIAS = <SYNONYM>                                140
                      | UUT_POINT = <UUT_POINTS> | <COMMENTS>
  6 <UUT_POINTS>  ::=  (<UUT_POINT_ID> [,<UUT_POINT_ID>]* )                  141
                     | <UUT_POINT_ID>
```

FIGURE 3.1 EBNF SPECIFICATION OF NOPAL (continued)

```
1       SPEC MINIRADIOSET;

         /*  TEST  MODULES  */

2       TEST DC_INPUT;   /* DC INPUT SHORT CHECK */
         /* NO STIMULI */
3         MEASUREMENT;
4           CONJUNCTION:  <J24_B, J24_C> = CONST_R(MRES OHM)
4                         TARGET:  MRES;
5           ASSERTION:  MRES > 100      SOURCE:  MRES;
6         LOGIC:  |¬ D2, *C3;
7             DIAGNOSIS D2: (INPUT_SHORT,, #4);
8             DIAGNOSIS D3:  (, (MRES, ' OHMS'),D);

9       TEST AMPL;
10        MEAS;
11          ASRT:   V1 = 0.26 +- C.06;
12        LOGIC:  *D7, |¬D8;
13            DIAG D7:  PARM=(V1, 'VRMS'), TYPE= D;
14            DIAG D8:  AFFECTED COMP=AMPL_TOL(STD_5MHZ_FREQ), TYPE= #6,
14                      OTHER PARM= 'AMPL';

15      TEST DISTORT_2W;
16        STIM;
17          CONJ:  SAME AS DCV_AMS;
18        MEAS;
19          CONJ:  <J19_L, GND> = DISTORTION(P1%, 1 KHZ)
19                    TARG:  P1;
20          ASRT:  P1 <= 5;
21        LOGIC:  *27, |¬30;
22            DIAG 30:  PARAM= (' 2W', 5.0), TYPE= #18, COMP=DISTORT(AUDIO_2W);
```

FIGURE 3.2 NOPAL SPECIFICATION MINIRADIOSET -- SAMPLE PROBLEM

```
23        TEST DISTORT_VOLT;
24          STIM  DCV_AMS;
25            CONJ:  SAME AS DCV EXCEPT
25                        J16 = SIGNAL_AM(25.002MHZ, +13DB, 0%, 1KHZ);
26          MEAS;
27            CONJ: <J19_A, GND> = SINE_D(V2 VOLT, *, 0 SEC)
27                          TARGET: V2;
28            ASSE  A1:  V2 >= 2.2;
29            ASRT  A2:  V2 <= 2.8;
30          LOGIC   *24, *25, |¬26;
31              DIAG 24:  TYPE=#15, TIME=0, RESPONSE=?;
32              DIAG 25:  PARMS=(V2, ' VAC'), TYPE = D;
33              DIAG 26:  TYPE=#17, COMP= REF_VOLT(AUDIO_10MW);

34        TEST FREQ;
35          STIM  DCV;   /* 27.5V DC TO PIN J24_B */
36            CONJ: <J24_B, GND> =  CONST_S(27.5 VOLT);
37          MEAS;
38            CONJ:  <J22, GND> = SINE_D(V1 VOLT, F1 HZ, VAR1 SEC)
38                          TARGET:  V1, F1;
39            ASRT:  IF VAR1 = 60 THEN  F1 = 5E6 +- 60

39                                 ELSE F1 = 5E6 +- 2.5;
40          LOGIC:  *D4, |¬D5, *D6;
41              DIAG  D4:  (,, #5, 0), VAR1;
42              DIAG D5:  (FREQ_TOL(STD_5MHZ_FREQ), 'FREQ', #6);
43              DIAG  D6: (, (F1, ' HZ'),D);

44        TEST DISTORT_10MW;
45          STIM:  SAME AS DCV_AMS;
46          MEAS;
47            CONJ: <J19_A, GND> = DISTORTION(P1 %, 2 KHZ)
47                        TARGET: P1;
48            ASRT:  P1 <= 3;
```

FIGURE 3.2 (continued)

```
49          LOGIC: *27, |-28;
50               DIAG 27:  PARMMETERS=(P1, '%'), TYPE=D;
51               DIAG 28:  COMP=DISTCRT(AUDIO_10MW),TYPE=#18,
51                         PARMS=(' 10MW', 1.0);

             /*  MESSGAE DEFINITIONS  */
52          MESSAGE 0:  ALIAS=DISLPAY, TEXT='$T: $P ';
53          MESS #4: 'R/T, DC INPUT SHORTED J24-B/J24-C ',
53                   'AN/GRC-106 DEFECTIVE.  CHECK PRINTOUTS FOR DEFECTS.',
53                   'PRESS STOP.';
54          MESS #5: 'IF A 12 MINUTE WARMUP IS DESIRED, KEY IN 720;'
54                   ' OTHERWISE, KEY IN 60.   PRESS YES.';
55          MESS  #6:   '$C DEFECTIVE.',
55                      '5.0 MHZ STD. OUT OF $P TOLERANCE.' ;
56          MESS  #15:  'MC & KC CONTROLS TO 250000.',
56                      'ADJUST AUDIO GAIN CONTROL FOR 2.2 TO 2.8 VAC',
56                      '(2.5 VAC NOMINAL).    PRESS  YES.';
57          MESS  #17:  '10 MW DISTORTION REFERENCE VOLTTAGE FAILED.';
58          MESS  #18:  '$P1 AUDIO DISTORTION GREATER THAN $P2 PERCENT.';

             /*  UUT CONNECTING POINTS  */
59          UUT_POINT:  J22, LIMIT=(VOLT, 70, 0, GND);
60          UUT_PT 2:  J24_B, ALIAS = XJ24_B, CONNECTOR=(MULTIPLE, B),
60                     LIMIT=(VOLT, 35, 20, GND);
61          UUT_P:    J24_C, ALIAS=GND, CONN=(MULTIPLE, C);
62          UUT_PT:     J16, CONNECTOR=COAXIAL, LIMIT=(UVOLT, 100, 0, GND),
62                                ' COAXIAL CABLE';
63          UUT_PT:    J19_A, LIMIT=(VOLT, 5, 0, GND);
64          UUT_PT:    J19_L, LIMIT=(VOLT, 70, 0, GND);
65          UUT_PT:     J19_B, ALIAS=GND;

             /*  UUT COMPONENT FAILURES  */
66          COMPONENT  1:   INPUT_SHORT;
```


FIGURE 3.2 (continued)

```
67        COMP_FAIL  2:  STD_5MHZ_FREQ, FAILURE FUNCTION= FREQ_TOL,
67                       INDEX= 1, PRCTECTION = 1;
68        COMP_FAIL 3:   STD_5MHZ_FREQ, FAIL FUNC = AMPL_TOL, INDEX = 2,
68                       PRCTECT = 1;
69        COMP  6:  AUDIO_10MW, FAIL = REF_VOLT, PROT=1, 'DISTORTION REF VOLT';
70        COMP_FL  7:  AUDIO_10MW, FAIL = DISTORT, PROTECTION=(1, 6);
71        COMP_FL:   AUDIO_2W, FAIL FUNC= DISTORT;

           /*      ATE FUNCTIONS */
72        FUNC 10:  CONST_S, FUNCTION TYPE =S, PARM= (X,S,LIMIT=(VOLT,60,0)),
72                            VALUE RETURNED =' CONSTANT VOLT.';
73        FUNC 20:  CONST_R, TYPE=M, PARAM= (X,T,(OHM,1000,1)),  VALUE='TRUE/FALSE';
74        FUNC 30:  SINE_D,ALIAS=SINE_DELAY, TYPE=M, PARM1= (X,T,(VOLT,10,-10)),
74            PARM2=(Y,T,(MHZ,10,0)), PARM2=(Z,S,SEC), 'APMPL., FREQ., TIME DELYD';
75        FUNCTION  40: DISTORTION, TYPE=M, PARM= (X,T,%),  /* % DIST. */
75                   PARM=(Y,S,(KHZ,100,0))  /* FREQ.  */,    VALUE='TRUE/FALSE';
76        FUNCTION 50: SIGNAL_AM, ALIAS=SAM, TYPE=S, #PINS = 1,
76                   PARM#1= (X,S,(MHZ,100,0.1)),  /* CRARIER FREQ */

76                   PARM#2= (Y,S,(DB,-10,-150) ),  /*   POWER */
76                   PARM#3= (Z, S, %),   /* MODULATION IN PERCENT  */
76                   PARM#4= (W, S, (KHZ,15,0.1))  /*  MOD FREQ.  */;
77        FUNC 110:  FREQ_TOL, TYPE=F, PARM=COMPONENT;
78        FUNC 120:  AMPL_TOL, TYPE=F, PARM=COMPONENT;
79        FUNC 130:  REF_VOLT, TYPE=F, PARM=COMPONENT;
80        FUNC 140: DISTORT, TYPE=F, PARM=COMPONENT;

           /*  ATE INTER-CONNECTING POINTS  */
81        ATE_POINT:  ATE_J24B, OUT_PT=J24_B;

82      END MINIRADIOSET;
```

FIGURE 3.2(continued)

in Appendix B. This sample problem will be referenced frequently in the subsequent sections of this chapter and also in later chapters.

This example describes a specification consisting of six tests on a radio set.

It begins with the specification of six tests (DC-input, amplitude, 2W-audio-distortion, distortion-reference-voltage, frequency, and 10w-audio-distortion) together with all the messages referred to in the tests. Then it identifies the UUT connecting points and component failures. Finally, the ATE functions and inter-connecting points are defined.

The numbers on the left-hand side are NOPAL statement numbers generated and printed by the NOPAL processor. Note that only one ATE inter-connecting point (ATE_J24B) is included in the specification. It is an over simplification intended for demonstration purpose. The set of real ATE inter-connecting points should be available and must be properly identified in the ATE specification section, when the UUT is actually connected to a real ATE for performing the tests.

3.2.3 Fundamental Constructs of NOPAL

Fundamental syntax units used in NOPAL, as shown in productions 1 through 52 of Figure 3.1, are briefly discussed in this section.

A string constant is either a character string or a bit string. Character strings are composed of any number of any characters enclosed in single quotes ('). If a single quote is part of the string, it is represented by two consecutive quotes. Bit strings are sequences of 0's and 1's enclosed in single quotes and immediately followed by the character B. For example, 'FREQ' and 'PRESS STOP'are character strings. '1'B is a bit string. String constants are primarily used as arguments of functions; character strings are also used to compose diagnosis messages.

A sequence of characters enclosed by /* and */ is a comment. For example, /* TEST MODULES */ is a comment. Comments are for the purposes of documentation only, and they are treated as blanks.

A number in NOPAL can be signed or unsigned, integer or floating point and may include an exponential part, such as 0.26, - 10, and 5E6.

An identifier is a sequence of alphanumeric characters, beginning with a letter. The twenty-six alphabets and four special characters (@, #, $, and _) are letters. For instance, DC_INPUT, V1, and #17 are identifiers. Identifiers are used as names for identifying certain entities.

A variable can be a simple variable which is an identifier, or a subscripted variable which consists of an identifier and a parenthesized subscript list. A subscript list is composed of one or more arithmetic expressions separated by commas. For example, MRES and V1 are variables. Variables are names which represent a data item that can take on different values during the execution of a program.

An arithmetic expression is a combination of numbers, variables and function calls connected by mathematical operators (+, -, *, /, and **) that yield a value when evaluated. A function call consists of an identifier possibly followed by a parenthesized list of arguments separated by commas. An argument is an arithmetic expression or a string constant. Function calls always return a value when invoked.

A relational expression consists of two arithmetic expressions separated by one of the relational operators (=, >, < , ¬=, < =, >=, etc.) which evaluate to either TRUE or FALSE logical value. A Boolean term is a logical expression consisting of the relational expressions which are connected by logical operators (NOT), |(OR), and

&(AND). An if-clause is a Boolean term enclosed by the two keywords IF and THEN, and is used to specify a conditional statement (conjunction or assertion). For example, "IF VAR1 = 60 THEN" is an if-clause, which includes a Boolean term (also a relational expression) "VAR1 = 60". Relational expressions are also used to form simple assertions (see Section 3.3.1.3).

A connector dimension expression, which is used to specify a set of UUT test points, is composed of a connector and optionally a dimension. A connector is a single UUT point name, or a list of UUT point names enclosed by angle brackets < >. Dimensions are physical units of measure, such as VOLT, AMP, SEC, etc. Thus < J19-A, GND > is a connector dimension expression.

A value dimension expression is an arithmetic expression possibly followed by a dimension, and is used as an argument of a stimulus or measurement function. For example, MRES OHM and 25.002MHZ are value dimension expressions.

A function dimension expression is a combination of function primaries, constants, and mathematical operators (+, -, *, and /). Its basic unit is function primary, which consists of a function identifier followed by a parenthesized list of functional arguments. A functional

argument can be a value dimension expression optionally prefixed by a relational operator, a range, a string constant, or an asterisk (*) meaning "don't care". A range is composed of two value dimension expressions separated by the operator +-, the first expression specifies nominal value and the second tolerance. Function dimension expressions are used to express waveform conjunctions in the stimuli or measurements (to be further discussed in Section 3.3.1). In the current version of the NOPAL processor, they have been implemented as function primaries only. Thus, CONST_R(MRES OHM) and SIGNAL_AM(25.002 MHZ, + 13DB, 0%, 1 KHZ) are function dimension expressions.

3.2.4 UUT Specification

The UUT-oriented information needed for the automatic test program generation is grouped into two sections: (1) UUT connecting points which identify the connecting pins for the testing and (2) component failures which define all the potential faulty components with failure modes (i.e. types of failures). This is used in conjunction with the test-modules specification (Section 3.3) for the purpose of identifying UUT connecting points and component failures, and for checking the correctness and completeness of the specified tests.

3.2.4.1 UUT Connecting Points

This section consists of a collection of UUT connecting points. The top level structure for a UUT connection point is shown in Figure 3.3(a), corresponding to productions 119-121 of Figure 3.1. It begins with the keyword UUT_POINT. Then an internal sequence number and a delimiter (:) may optionally follow. The internal sequence numbers, which are optional and have no semantics, enable the user to enumerate the set of UUT points in a certain sequence if he so desires. Next comes a symbolic name (identifier) to identify the connecting point. Finally one or more of the following keyword attributes may optionally appear:

(1) a synonym to the UUT point identifier. This name is used to identify a single point or a class of connecting points.

(2) connector type and point identification.

(3) protective limits which consist of dimension, maximum limit, minimum limit, and reference point enclosed in parentheses and separated by commas. This information is used to protect the connecting point from inadvertent damage caused by excessive stimuli power.

(a) UUT Connecting Point

```
< UUT_CONNECTION_POINT > ::=
        UUT_POINT  [< SEQ# >]  [:]< UUT_POINT_ID >
                   [,< UUT_POINT_KEYWORD >]*;
< UUT_POINT_KEYWORD >::=
          ALIAS = < SYNONYMN >
        | CONNECTOR = < UUT_CONNECTOR >
        | LIMIT  = < PROTECTIVE_LIMITS >
        | < COMMENTS >
```

(b) Example

```
UUT_POINT 2:  J24_B, ALIAS = XJ24_B,
        CONNECTOR = (MULTIPLE, B),
            LIMIT = (VOLT, 35, 20, GND);
```

FIGURE 3.3

TOP LEVEL STRUCTURE AND EXAMPLE OF UUT CONNECTION POINT

(4) Any comments enclosed in single quotes after the optional "COMMENT=".

Figure 3.3(b) illustrates a sample UUT connection point specification. The second UUT point, called J24-B or XJ24-B is connected to point B of a multiple connector. The protective limits are 35 volts for the maximum and 20 volts for the minimum with respect to another connecting point called GND.

3.2.4.2 UUT Component Failures

All the potential affected components (components with corresponding failure modes) must be listed in this section in order to identify affected components, to check completeness, and to produce fault isolation statistics. The top level syntax for each such entry is depicted in Figure 3.4(a). It consists of the following items in that order after the keyword COMP_FAIL:

(1) An optional sequence number to uniquely identify the component together with the failure node. This number may be referenced in the affected-components part of diagnoses, or in the component-protection (3-e) of this section:

(a) UUT Component Failure

```
<UUT_COMPONENT_FAILURE> ::=
        COMP_FAIL  [ <SEQ#> ] [:] <COMPONENT>
              [, <COMP_FAIL_KEYWORD> ]*;
<COMP_FAIL_KEYWORD>  ::= ALIAS = <SYNONYM>
          | FAILURE [FUNCTION] = <FAILURE_FUNCTION>
          | PARAMETER = <PARM_LIST>
          | INDEX = <FAILURE_INDEX>
          | PROTECTION = <COMPS_PROTECTED>
          | <COMMENTS>
```

(b) Example

```
COMP_FAIL 3:  STD_5MHZ_FREQ, FAIL FUNC = AMPL_TOL,
    INDEX = 2, PROTECTION = 1;
```

FIGURE 3.4 TOP LEVEL STRUCTURE AND EXAMPLE OF UUT COMPONENT FAILURE SPECIFICATION

(2) A symbolic name (identifier) for the component. The component is identified by this name or its synonym specified in 3(a).

(3) One or more of the following keyword attributes may be included optionally:

(a) A synonym to the component identifier.

(b) A failure function to indicate the type of failure for the component (e.g. Open, short, out-of-tolerance).

(c) Any other parameters of the failure function, if any.

(d) A failure index which is an integer used to index the components by their likelihood of failure. The smaller the index value, the larger the failure likelihood will be. This information could be used to obtain execution efficiency by first testing the components which are more likely to fail.

(e) Component protection - a list of other affected components whose failures will prohibit the testing of this component.

(f) Any comments enclosed in single quotes.

Figure 3.4(b) illustrates a sample UUT component failure specification. An affected component, numbered 3, consists of a component STD_5MH_FREQ and a failure function AMPL_TOL. It is given a failure index 2 and is protected by affected component 1 (which turns out to be component INPUT_SHORT).

3.2.5 ATE Specification

ATE related information which is needed to verify the test-modules and UUT specifications is organized in two sections: (1) ATE connecting points which are connected to the matching UUT connectors and (2) ATE functions specified in the stimuli and measurement sections of the test modules.

3.2.5.1 ATE Connecting Points

This section describes the ATE inter-connecting points which are interfaced with the matching UUT connecting points. The top level syntax of an ATE connecting point statement is shown in Figure 3.5(a). It consists of the following items in that order, after the keyword ATE-POINT:

(a) ATE Connecting Point

```
< ATE_CONNECTION_POINT > ::=
        ATE_POINT [ < SEQ# > ] [:] < ATE_POINT_ID >
            [, < ATE_POINT_KEYWORD > ]*;
< ATE_POINT_KEYWORD > ::=  ALIAS = < SYNONYM >
                         | UUT_POINT = < UUT_POINTS >
                         | < COMMENTS >
```

(b) Example

```
ATE_POINT:  ATE_J24B, UUT_POINT = J24_B;
```

FIGURE 3.5

TOP LEVEL STRUCTURE AND EXAMPLE OF ATE CONNECTING POINT

(1) an optional internal sequence number, which has no semantics. These numbers allow the user to enumerate the set of ATE points.

(2) a name (identifier) for the ATE connection point. The ATE point can be identified by this name or its synonym specified in 3(a).

(3) Optionally any of the following keyword attributes may appear:

(a) a synonym to the ATE point.

(b) a list of matching UUT connecting points to which the ATE point is connected.

(c) any comments enclosed in single quotes.

Figure 3.5(b) illustrates a sample ATE connecting point specification. The ATE connecting pointed is called ATE_J24B and is connected to a UUT connection point called J24-B.

3.2.5.2 ATE Functions

Functions used for the specification of stimuli, measurements, and component failures must be listed in this section. Purely computational functions, or iteration control functions, may also be used and listed here. The top level syntax of a function entry is depicted in Figure 3.6(a). It is composed of the following items in that order after the keyword FUNCTION:

(a) ATE Function

```
<ATE_FUNCTION>  ::=
        FUNCTION [ <SEQ#> ] [:]  <FUNCTION_ID>
            [,  <FUNCTION_KEYWORD> ]*;
<FUNCTION KEYWORD>  ::=  ALIAS = <SYNONYM>
        |  [FUNCTION] TYPE = <FUNCTION_TYPE>
        |  #PINS =  <UNSIGNED_INTEGER>
        |  PARAM =  <PARM> [, PARAM = <PARM> ]*
        |  VALUE [RETURNED] =  <VALUE_RETURNED>
        |  COOPERATION =  <COOP_FUNCTIONS>
        |  <COMMENTS>
```

(b) Example

```
FUNCTION 10:  CONST_S, FUNCTION TYPE = S,
     PARAM = (X,S, LIMIT =(VOLT, 60, 0)),
      VALUE RETURNED = 'CONSTANT VOLT.';
```

FIGURE 3.6

TOP LEVEL STRUCTURE AND EXAMPLE OF ATE FUNCTION SPECIFICATION

(1) An optional internal sequence number, which aids to enumerate the set of ATE functions.

(2) a function identifier, i.e., name of the function. Thus the function can be identified by this name or its synonym specified in 3(a).

(3) any of the following keyword attributes may optionally appear.

   (a) a synonym to the function identifier.

   (b) a function type which is S(stimuli), M (Measurement), F(Failures), E(Evaluation), or C(controls). E is the default.

   (c) The number of pins needed for a function of type S or M. 2 is the default.

   (d) A list of parameters, each may include a parameter name, parameter types (S for SOURCE, or T for TARGET), and parameter limits indicating dimension, maximum and minimum values.

   (e) a description of the value(s) returned from this function, enclosed in quotes. This serves as a comment about the value returned only; hence no semantics is associated with it.

(f) a list of cooperating functions which are needed for parallel execution with this function. This applies especially to the synchronization between cooperating stimuli and measurement functions.

(g) any comments enclosed in quotes

Figure 3.6(b) shows an example of ATE function specification. The function is given a sequence number 10 and is called CONST_S. It is a stimulus function and has one parameter of SOURCE type. This parameter should never exceed 60 volts or go below 0 volt. The function returns a constant voltage.

3.3 Test Modules Specification

This section includes a collection of test modules and the definitions of the diagnoses and messages which are referenced in any test module.

Its overall structure is summarized in Figure 3.7. The detailed syntax of the whole test-modules specification is included in productions 57 through 106 of Figure 3.1.

The test-modules specification is the core of the NOPAL specification and is most complex to prepare. To reduce the complexity, yet to provide the essential information, each test module is specified independently of the others. Thereby individual test modules can be

```
< TEST_MODULES_SPECIFICATION >::=
        [ < TEST_MODULES >]*   [ < DIAGNOSIS >]*
                               [ < MESSAGE >]*
< TEST_MODULE >::=   [ < STIMULI >]   [ < MEASUREMENT >]
                               [ < LOGIC >]
        < STIMULI >::=   [ < CONJUNCTION >]   [ < ASSERTION >]*
        < MEASUREMENT >::=   [ < CONJUNCTION >]
                             [ < ASSERTION >]*
        < LOGIC >::=   [ < OPERATOR > < DIAGNOSIS_ID >]*
  < DIAGNOSIS >::= < DIAGNOSIS ID >
    [ < OPERATOR MESSAGE >]   [ < OPERATOR RESPONSE > ]
<OPERATOR_MESSAGE>  ::=
    [ < FAILURE_IDS >]   [ < OTHER_DATA > ]
     < MESSAGE_ID > [  < TIMING >]
< MESSAGE >::= < MESSAGE_ID > < MESSAGE_TEXT >
```

FIGURE 3.7 TOP LEVEL STRUCTURE OF
TEST MODULES SPECIFICATION

modified, deleted, or added without affecting the rest of the test modules. The objective of a test module is to describe a single test which diagnoses some failures.

As depicted in Figure 3.7, a test module is composed of stimuli, measurements, and logic. They are discussed in Section 3.3.1. Diagnoses which are referenced in the logic parts of the test modules, are presented in Section 3.3.2. Finally the messages which are referred to in the diagnoses are covered in Section 3.3.3.

3.3.1 Test Modules

This section describes the three major components of a test module (in addition to the test module label): (1) the stimuli that need to be applied to the UUT at test time, (2) the measurements that need to be made with the comparisons that will determine the results, and (3) the logic that selects diagnoses based on the results.

Syntactically, a test module is fully described by the above-mentioned three major components (which are to be discussed in the subsequent sections), plus the following test-module header statement:

TEST [< TEST_LABEL >];

where the optional < TEST_LABEL >, which is either an identifier or unsigned integer, is used to identify the test module. For example, "TEST DC_INPUT;" is a test

header statement which identifies a test module called DC_INPUT.

3.3.1.1 Stimuli and Measurements

Stimuli and measurements have the same syntax structure. This common syntax is shown in Figure 3.8 (a and b). The only difference is the keyword (STIMULI versus MEASUREMENT). Each component is briefly discussed as follows:

(1) Keyword (STIMULI or MEASUREMENT) used to specify the respective type of entity.

(2) An optional label of the stimuli or measurement for identification purpose. It can be either an identifier or unsigned integer, and must be unique in the collection of all stimuli and measurements.

(3) An optional test label enclosed in parentheses used to identify the "parent" test module with which the stimuli or measurement is associated. This parental relationship ensures the pure non-proceduralness of the statements. If this field is not specified by the user, the NOPAL processor would assign by default the last test module as its parent.

(4) Waveforms which consist of one conjunction and/or one or more assertions (to be further discussed in Sections 3.3.1.2 and 3.3.1.3).

(a) Stimuli

```
STIMULI [ < STIMULI_LABEL >]  [( < TEST_LABEL >)];
        [ < CONJUNCTION >]  [ < ASSERTION >]*
```

(b) Measurements

```
MEASUREMENT [ < MEASUREMENT_LABEL >] [( < TEST_LABEL >)];
        [ < CONJUNCTION >]  [ < ASSERTION >]*
```

(c) Sample Stimuli Statement

```
STIMULI DCV_AMS (DISTORT_VOLT);
```

(d) Sample Measurement Statement

```
MEASUREMENT (DISTORT_VOLT);
```

FIGURE 3.8 SYNTAX STRUCTURES OF STIMULI AND MEASUREMENTS WITH EXAMPLES

Figure 3.8(c) illustrates a sample stimuli statement. The stimulus is called DCV-AMS and is contained in a test module DISTORT_VOLT. Figure 3.8(d) demonstrates a measurement statement. The measurement is a part of test module DISTORT_VOLT, but is not given a name.

3.3.1.2 Conjunctions

The conjunction part of the stimuli or measurement is used to describe UUT connecting points and the stimuli functions to be applied or the measurements to be performed. The basic form of this part is a conjunction of triplets (explained later), hence the name "conjunction" is adopted. The syntax is depicted in Figure 3.9. All the components after the keyword CONJUNCTION are briefly discussed in the subsequent paragraphs.

(1) An optional label (identifier or unsigned integer) for the conjunction which must be distinct in the set of all the conjunctions and assertions in the current test module.

(2) An optional parenthesized stimulus or measurement label used to identify the "parent" stimulus or

```
< CONJUNCTION > ::=
      CONJUNCTION [ < LABEL > ]  [( < PARENT > )];
         < CONJUNCTION_BODY > [ < DECLARATION > ]*;
< CONJUNCTION_BODY > ::= < TRIPLET_CONJUCT >
                       | < BACK_REFERENCE >
< TRIPLET_CONJUNCT > ::= < SIMPLE_CONJUNCTION >
                       | < IF_CONJUNCTION >
  < SIMPLE_CONJUNCTION > ::= < TRIPLET > [& < TRIPLET > ]*
     < TRIPLET > ::=
        [(] < CONN_DIM_EX > < RELATION >
                          < FUNC_DIM_EX > [)]
  < IF_CONJUNCTION > ::= < IF_CLAUSE > < SIMPLE_CONJUNCTION >
                [ELSE  < TRIPLE_CONJUNCT > ]
< BACK_REFERENCE > ::=  [SAME]  AS  < STIM_MEAS_LABEL >
                        [EXCEPT  < SIMPLE_CONJUNCTION > ]
< DECLARATION > ::= < VAR_TYPE > [:]  < VAR_LIST >
  < VAR_TYPE >  ::=  SOURCE | TARGET
  < VAR_LIST > ::=  [(] < VARIABLE > [, < VARIABLE > ]* [)]
```

FIGURE 3.9 SYNTAX STRUCTURE OF A CONJUNCTION

measurement with which the current conjunction is associated. If this field is not given by the user, then the NOPAL processor assumes by default the most recent stimulus or measurement as the parent of the conjunction.

(3) There are two ways of specifying the conjunction body, triplet conjunct or back-reference. A triplet conjunct is a simple or conditional conjunction of triplets. A triplet consists of three parts: (a) a connector dimension expression which specifies a set of UUT connection points where a waveform function is to be applied to or measured from, (b) a relation operator, equal sign =, and (c) a function dimension expression which describes the waveform function(s) to be applied or measured. Refer to section 3.2.3 for more details about these three parts. Triplets are the basic units of a conjunction. They are connected by a conjunction operator &, and each of them may optionally be enclosed in parentheses. Back-reference is a convenient way of referencing another conjunction section which has been or will be defined in a separate stimulus or measurement. It is a simple reference, or a reference with modification and/or addition of some triplets.

Note that only the conjunction part is affected and also that the reference is through the stimuli or measurement label rather than the conjunction label itself. For more information and some examples about back reference the reader is referred to Section 3.2.2.4 of [CHE 76], which was coauthored by this author and Mr. Che.

(4) Declaration of variables in the conjunction as SOURCE or TARGET. This information will be used by the NOPAL processor in determining the execution sequence of events. Basically, SOURCE variables are referenced here but are generated or evaluated elsewhere. TARGET variables are evaluated here and may be referred to elsewhere. Variables that are not explicitly declared will be considered to be SOURCE variables, by default.

Figure 3.10 illustrates three examples of unlabelled conjunctions. The first one is a measurement conjunction. It says that a resistance measurement function CONST_R is to be conducted between the two UUT connecting points J24-B and J24-C. The measured resistance, in units of ohms, is stored in a variable MRES, which is declared as TARGET. The second example shows a conjunction in the stimuli DCV. It indicates that a constant voltage of 27.5 volts is to be applied to UUT points J24-B and GND.

```
MEASUREMENT;
    CONJ:  <  J24_B, J24_C >= CONST_R(MRES OHM)
                                TARGET:  MRES;
STIMULI DCV;
    CONJ:  < J24_B, GND > =  CONST_S(27.5 VOLT);

STIM DCV_AMS;
    CONJ:  SAME AS DCV EXCEPT
        J16 = SIGNAL_AM (25.002MHZ, +13DB,
                         0%, 1KHZ);
```

FIGURE 3.10

EXAMPLES OF CONJUNCTIONS

The last example demonstrates a usage of the back reference in the conjunction part of the stimuli DCV_AMS. The conjunction part is the same as that of the stimuli DCV, except an amplitude modulated signal SIGNAL_AM is also to be applied to UUT point J16. Thus, the resultant conjunction part consists of two triplets, CONST_S applied to UUT points J24_B and GND and SIGNAL_AM applied to UUT point J16.

3.3.1.3 Assertions

The role of the assertions is to perform the pure computation needed to supplement the facilities of the conjunctions, and to determine the selection of the diagnoses. Unlike conjunctions, they do not include references to connection points, stimuli/measurement functions, or any dimensions. The syntax of assertion is shown in Figure 3.11. Each component after the keyword ASSERTION is briefly described in the following paragraphs.

```
< ASSERTION >::=  ASSERTION [ < LABEL >]  [( < PARENT >)]:
                  < ASSERTION_BODY >[ < DECLARATION >]*
< ASSERTION_BODY >::= < SIMPLE_ASSERTION >
                    | < IF_ASSERTION >
< SIMPLE_ASSERTION >::= < RELATIONAL_EXPR >
                      |   < RANGE_EXPR >
  < RELATIONAL_EXPR > ::=
          < ARITH_EXPR > < RELATION > < ARITH_EXPR >
  < RANGE_EXPR > ::=
  < ARITH_EXPR >= < ARITH_EXPR >+- < ARITH_EXPR > [%]
< IF_ASSERTION >::= < IF_CLAUSE > < SIMPLE_ASSERTION >
                      [ELSE  < ASSERTION_BODY > ]
```

FIGURE 3.11

SYNTAX STRUCTURE OF AN ASSERTION

measurement with which this assertion is associated. If it is omitted, then, by default, the most recent stimulus or measurement would become the parent of the assertion.

(3) There are two types of assertion body: simple assertion, or conditional assertion. A simple assertion has also two forms, a relational expression which consists of two arithmetic expressions separated by a relational operator as described in Section 3.2.3 or a range expression which is a relational expression with an equality relation, plus an expression for tolerance. The semantics of assertions are further explained at the end of this subsection. A conditional assertion is composed of if clauses, which are discussed in Section 3.2.3, and simple assertions.

(4) The declaration of variables, particularly TARGET type, in the assertion is required. As explained in Section 3.3.1.2(4), if a variable is not explicitly declared, it will be considered as a SOURCE variable, by default.

The range expression mentioned in the above item (3) is a convenient way of expressing a value falling within a certain interval. Each range expression of the form:

e1 = e2 +- e3 [%]

where e1, e2, and e3 are arithmetic expressions, will be

expanded into the following two relational expressions at code generation phase (Chapter 7):

e1 <= e2 + e3 [*e2/100] and

e2 >= e2 - e3 [*e2/100].

where the square-bracketed expressions appear if the percent (%) has been included in the original range expression. In this case, e3 denotes a percentage with respect to e2 rather than an absolute value.

To be precise, there are two classes of simple assertions in the forms of relational expressions: (a) general relational expressions of the form e1 op e2, not involving any TARGET variables, or (b) special relational expressions of the form X = e2; where e1 and e2 are arithmetic expressions, op is a relational operator, and X is a TARGET variable. In case (a), the two expressions e1 and e2 are first evaluated and compared, and then the relational expression yields a TRUE or FALSE logic value depending on the two evaluated values and the operator op. In case (b), the arithmetic expression e2 is first evaluated, and then the result is assigned to the variable X. Thus the equal sign is interpreted as an assignment operator.

Stimuli and measurement assertions, while using the same syntax, are interpreted differently by the NOPAL system. The stimuli assertions are interpreted to

generate data as in case (b) only. An assertion in the measurement section can be used either to generate some data as in case (b) or to describe some condition as in case (a). To simplify the description of the semantics, each measurement assertion that generates data may be considered also to return a logical value TRUE. Thereby, the conjunctive value of all the measurement assertions and conjunction is considered to be the resulting value of the test module. Then the logical operators (see Section 3.3.1.4) are used to select appropriate diagnoses based on this result (possibly in conjunction with the results of other test modules).

Figure 3.12 illustrates two examples of assertion statements. The first one is a simple assertion. It specifies that the value of a SOURCE variable MRES should be greater than 100. The second example demonstrates a conditional assertion. Two SOURCE variables VAR1 and F1 are involved. If VAR1 is equal to 60, then F1 should lie between (5,000,000-60) and (5,000,000 + 60). Otherwise F1 should be between (5,000,000-2.5) and (5,000,000 + 2.5).

```
ASSERTION:  MRES > 100
                SOURCE:  MRES;

ASRT:  IF VAR1 = 60 THEN F1 = 5E6 +- 60
                    ELSE F1 = 5E6 +- 2.5;
```

FIGURE 3.12 EXAMPLES OF ASSERTIONS

3.3.1.4 Logic Parts

The logic part of the test modules provides several functions: (1) selects diagnoses based on the logical values returned by test modules, (2) facilitates sharing of the diagnoses among test modules, and (3) facilitates interactive communication with the ATE operator.

The syntax of a logic statement is shown in Figure 3.13(a). It consists of a list of pairs of logic operators and diagnosis labels, in addition to the key-word LOGIC and an optional label. There are two types of logic operators: (1) logic connectives (|, |¬ , &, &¬ , *) by which a test module (i.e., a stimuli-measurement pair) selects a diagnosis and (2) "after" operators (?,?¬) by which a diagnosis selects a test module. They are further explained below.

An "or" operator (|) is used to indicate that the corresponding diagnosis will be selected when the test module returns a TRUE value. An "or-not" operator (|¬) indicates that the diagnosis will be selected if the test module returns a FALSE value.

The two conjunction operators (&, &¬) are used when two or more test modules are required to select a diagnosis conjunctively. An "and" operator (&) indicates that a TRUE value returned by the test module is a necessary condition for selecting the corresponding diagnosis. Similarly, an "and-not" operator (&¬)

(a) Syntax

```
< LOGIC >::= LOGIC [ < LABEL >]  [:]
                [ < LOGIC_OPERATOR > < DIAGNOSIS_ID >]*
 <  LOGIC_OPERATOR >::= < LOGIC_CONNECTIVE >
                    [ < AFTER >] | < AFTER >
   < LOGIC_CONNECTIVE >::=   ||¬|& |&¬ | *
    < AFTER > ::= ?|?¬
< DIAGNOSIS_ID >::= < LABEL >
```

(b) Example

```
LOGIC:  |¬ D2, *D3, &100;
```

FIGURE 3.13 SYNTAX STRUCTURE AND EXAMPLE OF A LOGIC STATEMENT

indicates that a FALSE test result is necessary for selecting the diagnosis.

The "don't-care" operator (*) indicates that the specified diagnosis will be selected no matter what the test result is.

The two "after" operators (?, ?¬) are used to accommodate interactive communications between the ATE computer and the operator. An "after" operator (?) indicates that the test module is to be executed immediately after the diagnosis is selected and if the operator responds "YES" (Y). The "after-not" operator (?¬) exactly does the same thing except that the operator responds "NO"(N). Operator response is further discussed in Section 3.3.2.

Figure 3.13(b) illustrates an example of logic statement. Three diagnoses (D2,D3, and 100) and three corresponding logical operators (|¬, *, and &) are involved. If test result is FALSE diagnosis D2 will be selected. It is necessary (but not sufficient) that the current test result is TRUE in order to select diagnosis 100. Diagnosis D3 will always be selected no matter what the test result is.

3.3.2 Diagnoses

This section describes the diagnoses which are referenced in the logic parts of the test modules. The overall syntax structure of a diagnosis is depicted in Figure 3.14. As indicated, the diagnosis body, after the keyword DIAGNOSIS and diagnosis label, can be expressed in one of the two forms: positional or keyword form. Each field in the positional form must be specified in its proper position. Fields in the keyword form can be specified in any order, but each of them must be prefixed with a corresponding keywords and an equal sign (=). The diagnosis body is composed of two major components: (1) operator message and (2) operator response, each is briefly discussed below.

The operator message component describes the set of affected components and the message to be sent. It is further broken into the following four parts:

(1) Affected components - a list of the component failures that this diagnosis asserts. They are connected either by a disjunction operator (|) or conjunction operator (&), but not mixed. Each affected component can be specified either by the failure function followed by the parenthesized component explicitly, or by the

```
< DIAGNOSIS >::= DIAGNOSIS  < LABEL >[:]  < DIAG_BODY >;

  < DIAG_BODY >::= < POSITIONAL_DIAG >| < KEYWORD_DIAG >

< POSITIONAL_DIAG >  ::= [ < OPERATOR_MESSAGE >]
                              [, < OPERATOR_RESPONSE >]

< OPERATOR MESSAGE >::= ([ < AFFECTED_COMPONENTS >]
          [,[ < OTHER_PARAMETERS >]   [,[ < TYPE >]
                                      [, < TIMING >]]])

< OPERATOR_RESPONSE >::=  ?  |
          [(] < VARIABLE >[, < VARIABLE >]*[)][,] [?]

< KEYWORD_DIAG >::=  [OPERATOR MESSAGE:]
                        < DIAG_KEYWD >  [, < DIAG_KEYWD >]*

< DIAG_KEYWD >  ::=
      [AFFECTED] COMPONENT = < AFFECTED_COMPONENTS >
    | [OTHER] PARAMETER =  < OTHER_PARAMETERS >
    |  TYPE =  < TYPE >
    |  TIME =  < TIMING >
    |  RESPONSE =  < OPERATOR_RESPONSE >
```

FIGURE 3.14
SYNTAX STRUCTURE OF A DIAGNOSIS DEFINITION

corresponding component-failure sequence number (see Section 3.2.4.2). Also, a failure function followed by a list of components, F(C1 op C2 ... op Cn), is an abbreviated form of F(C1) op F(C2) ... op F(Cn), where F is a failure function, op a disjunction or conjunction operator, and Ci are component identifiers.

(2) Other parameters which will be inserted in the diagnosis message. Each parameter is a variable, number, or string constant.

(3) Message type which identifies the diagnosis message to be sent (see Section 3.3.3).

(4) Timing which indicates when the message is to be sent with respect to the beginning of the stimuli application. If omitted, it means that the message will be sent upon conclusion of the test module.

The operator response component specifies the instructions to the operator to perform some duties. It consists the following two parts:

(1) Y/N (represented by ?) response from the operator. The operator instructs to proceed with the suspended test or to initiate a subsequent test by responding Y(YES). Otherwise, he wants to discontinue

the normal testing by responding N(NO). In this case, a subsequent test connected by the "after-not" operator (?¬) may be initiated next.

(2) A list of variables into which the operator is requested to enter a corresponding list of values, and to key in Y to proceed.

Figure 3.15 illustrates an example of diagnosis specification, (a) in positional form and (b) in keyword form. The diagnosis is called D8. It is composed of three parts: an affected component, a character string parameter 'AMPL' and a message type #6. The affected component consists of failure function AMPL_TOL and UUT component STD_5MHZ_FREQ.

3.3.3 Messages

Messages can be shared by a number of diagnoses. A few message types are considered to be adequate since their actual texts can be modified by inserting parameters at execution time.

Figure 3.16(a) shows the syntax of a message definition. It consists of three fields after the keyword MESSAGE: (1) a label (an identifier or unsigned integer) used to identify the message type, (2) an optional synonym to the message label after the keyword ALIAS and equal sign, and (3) message text which is

(a) Positional Form

```
DIAGNOSIS D8:
        (AMP_TOL(STD_5MHZ_FREQ), 'AMPL', #6);
```

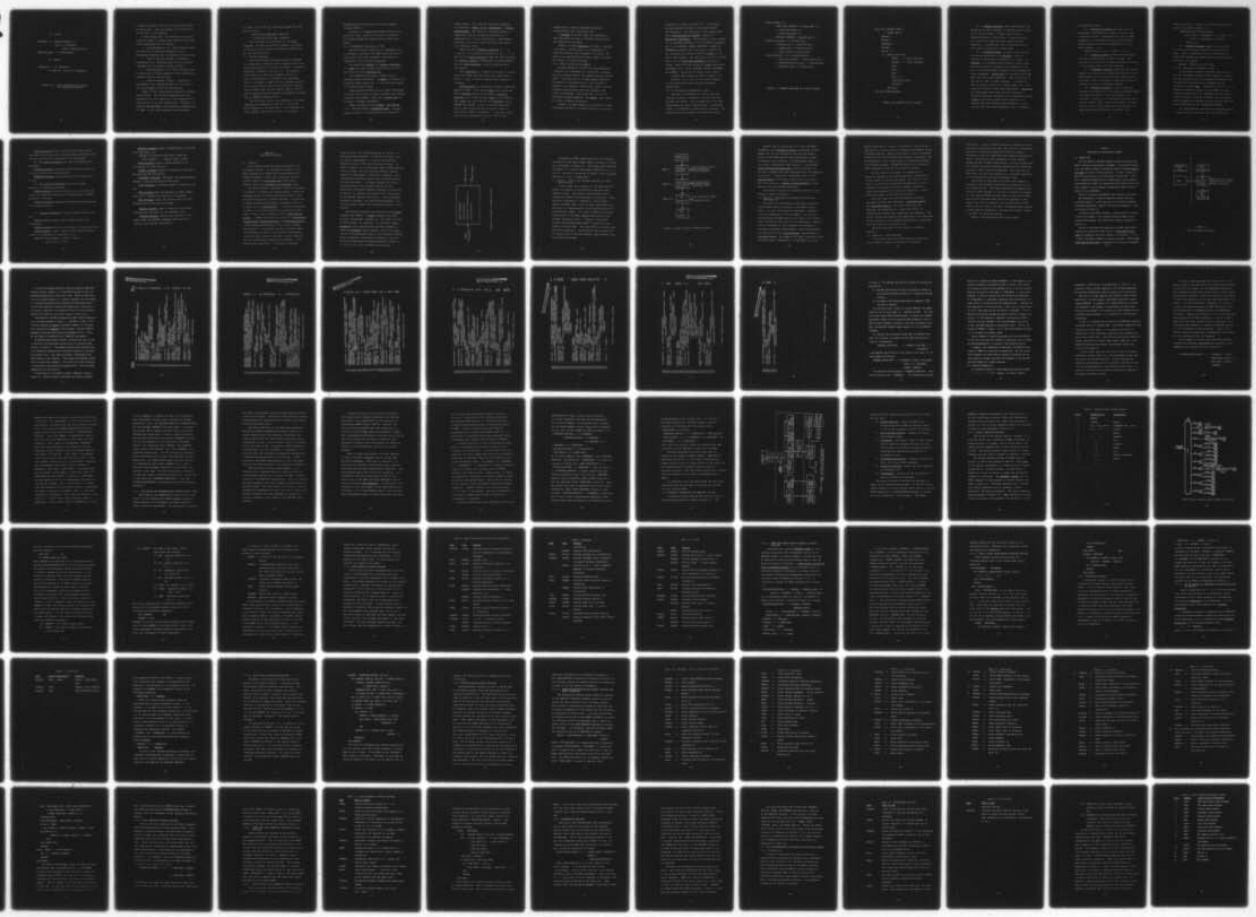
(b) Keyword Form

```
DIAGNOSIS D8:
        COMP = AMPL_TOL(STD_5MHZ_FREQ),
        TYPE = #6,
        PARAM = 'AMPL';
```

FIGURE 3.15 EXAMPLE OF DIAGNOSIS SPECIFICATION

AD-A054 910 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 14/2
AUTOMATIC TEST PROGRAM GENERATION.(U)
MAR 78 Y K CHANG DAAA25-75-C-0650
UNCLASSIFIED 77-02 ECOM-75-0650-F-1 NL

2 OF 4
AD
A054910



(a) Syntax

```
<MESSAGE> ::=  MESSAGE <LABEL> [:]
               [ALIAS =  <SYNONYM> ]
                      [TEXT=] <MESSAGE_TEXT> ;
<MESSAGE_TEXT>  ::= <CHAR_STRING>
```

(b) Example

```
MESSAGE #6:  '(C) DEFECTIVE',
             '5.0 MHZ STD. OUT OF (P) TOLERANCE';
```

FIGURE 3.16 SYNTAX STRUCTURE AND EXAMPLE OF A MESSAGE DEFINITION

a character string, after the optional keyword TEXT and equal sign. Thus the message can be identified by the label or its synonym.

In the message text, the following four special items may appear to indicate the locations where some actual parameters will be inserted.

1. A parenthesized C, (C), which indicates that the whole set of affected components in the referencing diagnosis will be inserted literally here.

2. C and an unsigned integer i enclosed in parentheses (Ci), indicating that the i-th affected component will be inserted in this position.

3. A parenthesized P, (P), which indicates that the whole set of other-parameters in the referencing diagnosis will be inserted here.

4. P and an unsigned integer i enclosed in parentheses, (Pi), indicating that the i-th item of the other-parameters will be inserted here.

Figure 3.16(b) illustrates an example of message specification. The affected components and other parameters are to be inserted in the message text as indicated by (C) and (P) respectively. Therefore, if AMPL_TOL(STD_5MHZ_FREQ) is the affected component and if 'AMPL' is the other parameter (as illustrated

in Figure 3.15), then the resulting message text will be as follows:

AMPL_TOL(STD_5MHZ_FREQ) DEFECTIVE

5.0 MHZ STD. OUT OF AMPL TOLERANCE.

This concludes the description of the NOPAL language. For more illustrative examples, the reader is referred to the separate documentation [CHE 76] by Mr. Che and this author.

3.4 The OPAL Language

OPAL (Operational Performance Analysis Language) is a high-level, procedural test language in which one can program tests for a variety of devices - electronic, mechanical, hydraulic, optical, etc. The tests for units under test (UUT) are designed to be run on various configuration of automatic test equipments (ATE). Its design is based upon the language contructs and concepts derived from the existing test languages such as ATLAS, GOAL, and CTL [ARI 73, NAS 73, WAR 74], and the general purpose high-level programming languages such as FORTRAN, ALGOL and PL/1.

The definition, syntax, and semantics of the OPAL language are documented in [FRA 76]. Since some minor changes of, and additions to, this language have been underway, another more recent, but unedited

documentation may be referred to for the updated features [FRA 77].

Section 3.4.1 summarizes the basic lexical constructs of OPAL. Section 3.4.2 gives an overview of an OPAL program and various types of OPAL statements.

3.4.1 Fundamental Constructs of OPAL

The basic lexical constructs or components such as constants, operators, and names which are used to form higher level OPAL statements and program are summarized in the following paragraphs.

The twenty-six letters, ten digits, break sign (-), blank and other special characters such as +, -, and $ form the OPAL character set.

For documentation purpose, comments may be included between statements. A comment is sequence of characters which begins with //, and ends with another // or end-of-line of input.

Some punctuation marks such as comma, parentheses, colon and dollar sign are used to separate one language construct from another. Note that a dollar sign ($) is used as statement end-marker.

NOPAL constants include integer, real, Boolean (i.e., TRUE and FALSE), character-string, and bit-string constants. String constants are enclosed in

single quotes. Four types of bit-string constants are supported: Binary, Octal, Hexadecimal, or Binary coded-decimal. Note that in a character string, a single quote and an exclamation point must be represented as !' and !! respectively. Some editing characters such as tab, page, and space may appear in a character string.

OPAL uses five arithmetic operators (+, -, *, /, and **), six relational operators (=, #=, < ,>, <=, and >=), five logical operators (AND, OR, NOT, and XOR), two rotation operators (ROTATE_LEFT and ROTATE_RIGHT), and four shift operators (logical/arithmetic shift left/right).

OPAL identifiers (or names) are strings of letters, digits and break characters which must begin with a letter. A name is used to identify some construct in a program.

Reserved-words are identifiers having a predefined meaning in OPAL. They include verbs such as SET and APPLY, keywords such as BY and RESULT, built-in-functions such as ABS and MAX, units such as AMP and VOLT, nouns such as AC and DC, and modifiers such as CURRENT and VOLTAGE. A unit is an OPAL identifier used to name a physical unit of measure (which is equivalent to NOPAL's dimension). A noun and a modifier are both identifiers which are used to name,

respectively, a class of measureable physical characteristics and a particular one.

A variable is an OPAL name that represents a data item which can take on different values during the execution of a program.

Finally, an OPAL expression is either a constant, a variable, a function call, or a combination of these and operators that evaluates to a value. The value of an expression may be real, integer, boolean, bit-string, or character-string.

3.4.2 OPAL Program and Statements

An OPAL program is an OPAL text which describes a complete test procedure for a given UUT, including the data transfer between the computer and the outside world. The first part of an OPAL program names and describes the characteristics of the ATE which is assumed to be available, the UUT test points, and the variables and procedures which are global to the entire test program. The second part is a set of executable program segments, call blocks, that consist of series of smaller procedures.

Formally, an OPAL program is a collection of OPAL statements enclosed by a program header and a program

terminator, as shown in Figure 3.17. An optional program name may be given in the program header and terminator. The first part of the program body, called main-descriptive section, consists of descriptive statements. Specifically it must contain: (1) all REQUIRE statements, (2) all SPECIFY statements and (3) the DECLARE, DEFINE, PARTITION, COMMON and REFERENCE statements which are used to describe all the variables and procedures that are global to the entire OPAL program. The remaining part of the program body is set of executable statements contained in blocks. Each block may include all local descriptive statements (i.e., except *REQUIRE*, *SPECIFY*, *REFERENCE* and COMMON). Blocks may be nested within blocks.

A sample OPAL program is shown in Figure 3.18.

Sections 3.4.2.1 and 3.4.2.2 briefly describe all the descriptive and executable statements respectively.

3.4.2.1 Descriptive Statements of OPAL

Descriptive statements are used to name and describe variables, procedures, virtual resources, and UUT test points. All the names used in an OPAL program must be either reserved-words, or statement labels, or described in a descriptive statement. The seven descriptive statements are summarized below.

```
< OPAL_program >::=
        BEGIN OPAL PROGRAM [ < program_name >]$
        [ < descriptive_stmt >]*
        [ < executable_stmt >]*
        END OPAL PROGRAM [ < program_name >]$
< descriptive_stmt >::= < declare_stmt >
        | < define_stmt >| < partition_stmt >
        | < common_stmt >| < reference_stmt >
        | < require_stmt>| < specify_stmt >
< executable_stmt >::= < calculation_stmt >
        | < input_output_stmt> | < flow_control_stmt >
        | < subroutine_control_stmt> | < tasking_stmt>
        | < interrupt_stmt >| < testing_stmt >
```

FIGURE 3.17 OVERALL STRUCTURE OF AN OPAL PROGRAM

```
BEGIN OPAL PROGRAM DUMMY  $
        //  DUMMY TESTS $
      REQUIRE ...
      SPECIFY...
      REFERENCE...
      DECLARE ...
      DEFINE ...
      L1:  BEGIN_BLOCK BLK1$
                DEFINE ... // LOCAL ROUTINES
                DECLARE ...// LOCAL VARIABLES
                APPLY ...
                DELAY ...
                READ  ...
                SET X = ...
                BEGIN_BLOCK BLK2$
                END BLK2$
           END BLK1$
END OPAL PROGRAM DUMMY$
```

FIGURE 3.18 EXAMPLE OF OPAL PROGRAM

(1) A DECLARE statement names and describes the properties of OPAL variables. A variable must be given one of the five modes: Integer, Real, Boolean, Character-string, or Bit-string. It may be declared as an array with upper and lower bounds for each dimension. A variable may also be associated with a unit, and/or initialized to some value.

(2) A DEFINE statement is used to give the properties and definition of a procedure: either a function, a subroutine, an interrupt, or a task. A function is a procedure which returns a single value when the function is invoked in an expression by the use of the function-name followed by the parenthesized actual arguments. Subroutines are procedures which may have input and output parameters. They are invoked by the use of a CALL or TABLE statement. A subroutine does not return a value as a function does, but it may change the values of some output parameters. Interrupts are subprograms which check for the occurrence of critical conditions and define the corresponding actions. A task is a procedure which can be carried out in parallel with the invoking program segment. It may include input and output parameters, and is invoked by

an ACTIVATE statement.

(3) A PARTITION statement describes a set of range-names, associated with ranges of values, that a variable can have. Range-names are used in TABLE statements, or IS expressions (which yield TRUE or FALSE depending on whether expressions are in the given ranges); e.g.,

PARTITION X AS LO < = 3 < NOM < 8 < HI $

(4) A COMMON statement defines those variables which are shared between two OPAL programs. The statement must appear in the main-descriptive sections of the programs which use these variables.

(5) A REFERENCE statement identifies the program segments or files which are described external to the OPAL program, but are used within the program being processed. This is intended to facilitate the use of program libraries or independent modules.

(6) A REQUIRE statement is used to name and describe the properties of the virtual resources needed to perform the test of a UUT, and assumed to be available. Basically, it includes the name and type (source, sensor, load, clock, input device, or output device) of each virtual resource, and optionally the specifications of limitations, accuracies, and

connection points. It must be in the main-descriptive section of an OPAL program; e.g.,

```
REQUIRE ACS AC VOLTAGE SOURCE
    WITH 0 AMP <= CURRENT_AV <= 10 AMP +- 5PC,
       -10 VOLT <= VOLTAGE_AV <= 10 VOLT
    CNX HI, LO$
```

(7) A SPECIFY statement names the test points on a UUT, and optionally imposes the constraints on them. A UUT is modelled as a set of test-points by using these SPECIFY statements in the main-descriptive section of the program.

Example: SPECIFY TP1, TP2, TP3$

3.4.2.2 Executable Statements of OPAL

An executable statement describes some activity that the programmer wishes the ATE computer to perform. All the statements which are referenced elsewhere must be labelled by preceding each statement with an identifier, called label, followed by a colon (:). A labelled statement internal to a procedure or block cannot be referenced by any statement outside the procedure or block. As shown in Figure 3.17, the executable statements are grouped into seven types, which are briefly described in the following paragraphs.

(1) SET statement calculates the expression on the right-hand side of the equal sign, and assigns the

resultant value to the variable on the left-hand side. Example:

```
SET C = SQRT(A*A + B*B) VOLT$
```

(2) Input-output statements consist of an OUTPUT statement which causes a transfer of data to an output device, and INPUT statement which causes a transfer of data from an input resource. Optionally, both statements may include a format expression.

(3) Flow-of-control statements are used to control the execution sequence of events through an OPAL program.

IF statement causes execution one of the two groups of statements depending on the value of a condition, using the conventional IF_THEN_ELSE construct.

REPEAT statement causes repetitive execution of a sequence of statements until a termination condition has been satisfied.

LEAVE statement causes the sequential execution of a IF, REPEAT, ESCAPE, DO, or TABLE statement to stop, and then to resume execution at the next statement following the indicated construct.

CYCLE statement causes sequential execution of a REPEAT statement to stop, and to resume at the cycle point of the REPEAT statement.

GOTO statement causes to transfer unconditionally to the designated statement.

TABLE statement implements a decision table or a two-dimensional case statement, by selecting a set of actions from several alternatives based on a set of conditions; e.g.,

```
TABLE                          T          V;
        BROKEN_FAN             HI         LO;
        DEFECTIVE_PUMP         LO         HI;
END TABLE$
```

DELAY statement delays the execution of next statement for either a given time or until manual intervention.

INCLUDE statement causes the text of the named external program segment to be incorporated in line into the program.

BEGIN BLOCK statement delimits a block of OPAL statements, which may contain local descriptions other than COMMON, REFERENCE, REQUIRE, or SPECIFY statements.

CHAIN statement causes linking of the current OPAL program to the named OPAL object program, with optional starting location of execution.

(4) Subroutine-control statements are used to manage the execution of subroutine procedures:

CALL statement causes the named subroutine to be invoked, possibly with arguments to be passed to the corresponding parameters.

RETURN statement returns control to the procedure that called the subroutine.

ESCAPE statement returns control to the escape action contained in the current CALL statement when an abnormal condition has occurred in the execution of the subroutine.

(5) Two tasking statements are used in managing task procedures:

ACTIVATE statement initiates execution of the named task in parallel with the current procedure.

TERMINATE statement causes execution of the named task to stop.

(6) Two interrupt statements are used to manage execution of interrupt procedures:

ENABLE statement assigns a priority to an interrupt and causes the interrupt condition to be checked until a DISABLE statement is executed.

DISABLE statement suppresses the detection and processing of the named interrupt.

(7) Testing statements are used to perform tests on the UUT:

START statement causes a time reference point to be established.

CHANGE statement modifies the setting of a virtual resource.

APPLY statement causes a resource with indicated limitations to be connected to a test-place; e.g.,

```
APPLY ACS WITH VOLTAGE = 7 VOLTS, +-10 PC
              AT HI = P1, LO = P2$
```

MEASURE statement causes a characteristic of the UUT to be measured; e.g.,

```
MEASURE AC VOLTAGE IN VOLT INTO X USING ACS
   WITH VOLTAGE < = .7 VOLT AT HI=P1, LO=P2$
```

SETUP statement causes a named resource to be initialized in a specified way.

CONNECT statement connects a resource to the UUT at the specified *UUT test-points*.

DISCONNECT statement eliminates the interconnection between a resource and the UUT test-points.

CLOSE statement initiates or gates a resource to the UUT.

OPEN statement does the opposite of CLOSE, namely, turns off or inhibits the function of a resource.

READ statement causes the current value of a UUT characteristic to be read and stored.

REMOVE statement causes a resource to be opened, disconnected, and reset to its quiescent state.

MONITOR statement combines the functions of the APPLY, READ, and OUTPUT statements which are to be repeated until manual intervention.

CHAPTER 4
THE NOPAL PROCESSOR

4.1 Overview

This chapter covers the overall description of the NOPAL Processor. As presented in Chapter 3, the NOPAL language is a higher-level non-procedural programming language in which one can specify tests on various classes of units-under-test (UUT) under a computer-controlled automatic test equipment (ATE).

The NOPAL processor (hereafter called the Processor) is designed to automate the program design, coding, and debugging phases of program development based on test specifications written in NOPAL language. A collection of NOPAL statements describing a functional module is referred to as a specification. A complete description, in NOPAL, of the tests desired for a given UUT under a given "virtual" ATE is referred to as a NOPAL specification. A NOPAL specification consists of test-modules-specification, UUT-specification, and ATE-specification. A test-modules-specification is the core of a NOPAL specification. It describes the set of desired tests by specifying the stimuli to be applied, measurements to be performed, logics to be used to select diagnoses, and the corresponding diagnoses and messages. A UUT-specification identifies: 1) potential component

failures and 2) the connecting points of the UUT. An ATE-specification defines: 1) the ATE functions, such as stimuli, measurements, or special computational capabilities and 2) ATE inter-connecting points. As shown in Figure 4.1, the Processor takes a NOPAL specification as input, and then performs analyses (syntax, semantics, completeness, consistency, ambiguity, etc.), test-modules sequencing (intra- and inter-test-module) and code generation. It finally produces as output a procedural object test program (written in OPAL, Operational Performance Analysis Language, as explained in Chapter 3), together with various user reports such as reformatted specification listing, cross-references, sequencing flowchart, and error/warning messages. All of these output reports will be fully described in later chapters.

The processor performs the translation from source language (here NOPAL) to target language (here OPAL) as other conventional compilers do. Two other functions of the Processor are important. It processes a non-procedural source specification and then generates a complete procedural target program, based on an application of directed-graph theory. The Processor provides better system-user interaction by sending to the user proper warnings/errors indicating necessary changes or additions to the submitted NOPAL specification.

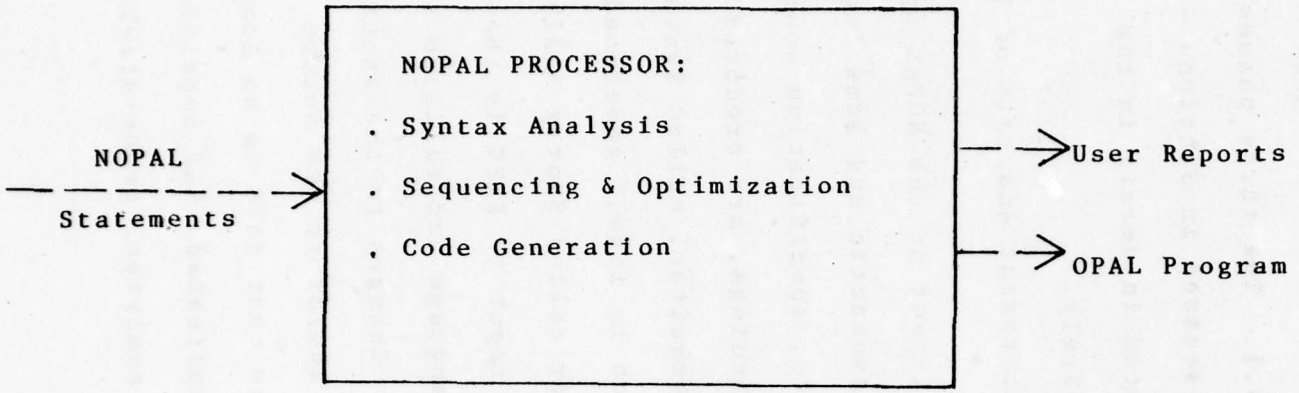


Figure 4.1 Overview of NOPAL Processor

Processing of NOPAL specification by the Processor consists of three major phases shown in Figure 4.2, which is a refinement of Figure 4.1. The three phases depicted in Figure 4.2 is briefly discussed in Sections 4.2, 4.3 and 4.4, and will be presented in detail in the following Chapters 5, 6 and 7 respectively.

4.2 Phase I: Syntax and Statement Analysis of NOPAL Specification

In this phase each statement of the NOPAL specification is analyzed to find syntactic and some local semantic errors. Two reports, specification source listing and syntax errors/warnings, are produced. These tasks are performed by a submonitor, called Syntax Analysis Program (SAP), which is itself generated automatically by a meta-processor called Syntax Analyis Program Generator (SAPG). The input to SAPG is the formal syntax rules of the NOPAL language and calls on some routines (written in PL/I). Changes to the syntax of NOPAL during development and/or in the future can thereby be easily made. Note that SAPG is no longer used once SAP is generated. As indicated, SAP consists also of routines such as lexical analyzer, error-stacker, and store/retrieve package.

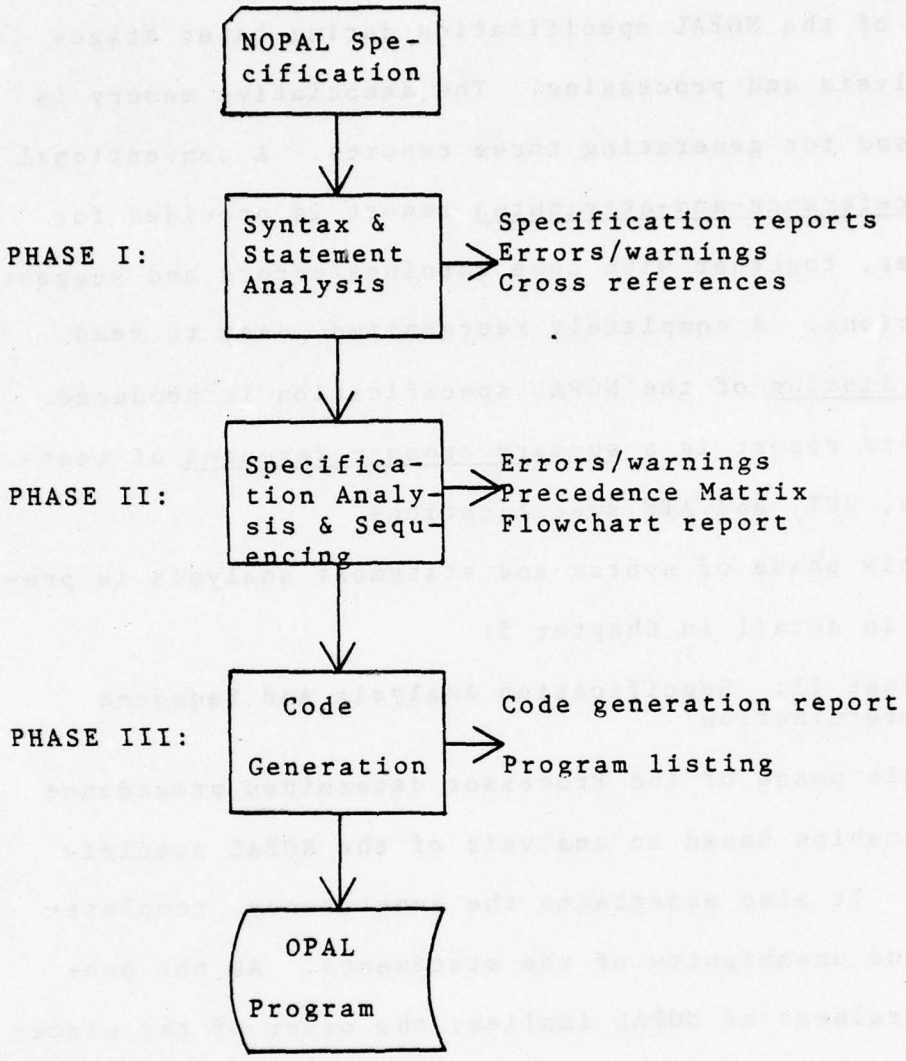


Figure 4.2 Major Phases of NOPAL Processor

Another task of this phase is to store the NOPAL statements in an associative memory (stimulated on main memory) for ease in subsequent retrieval and modification of the NOPAL specification during later stages of analysis and processing. The associative memory is also used for generating three reports. A conventional cross-reference-and-attributes report is provided for the user, together with some warnings/errors and suggested corrections. A completely reorganized, easy to read source listing of the NOPAL specification is produced. The third report is a summary cross-references of test-modules, UUT, and ATE specifications.

This phase of syntax and statement analysis is presented in detail in Chapter 5.

4.3 Phase II: Specification Analysis And Sequence Determination

This phase of the Processor determines precedence relationships based on analysis of the NOPAL specification. It also ascertains the consistency, completeness, and unambiguity of the statements. As the non-proceduralness of NOPAL implies, the order of the statements provided by the user is of no consequence. However, various components of the statements are analyzed to determine precedence relationships. These relationships are then used to form a directed graph, represented by a precedence matrix. Each node of the graph represents a data (variable) name, a diagnosis, or a statement (or a test

module consisting of a group of statements). Each directed edge denotes a certain type of precedence relationship, having a related priority. Based on the graph, it can be determined whether the test specifications are complete, consistent, and/or unambiguous. Also, a user report is produced containing error/warning messages, assumptions made by the Processors, and/or appropriate actions to be taken by the user.

The next task in this phase is to determine the execution sequence of all events implied by the specification, based on the directed graph. The result of this task is a set of data structures representing a correct sequence of processes and flow of events, assigned to levels and sequenced in the order of execution. A flow-chart-like report is produced for the user.

Note that there are two sub-phases of the above mentioned analysis and sequencing: intra-test-module and inter-test-module. The former deals with the analysis, and sequencing of a given test module, by examining the conjunctions, assertions, and diagnoses. The latter concerns the analysis and sequencing of the collection of test modules in a given NOPAL specification, considering each test module as an integral unit.

Detailed description of this phase is covered in Chapter 6.

4.4 Phase III: Code Generation

In this phase the object OPAL program is generated. First, global variables are declared and properly

initialized. Second, an OPAL subroutine is defined for every test module specified by the user; the code for a test-module subroutine is generated immediately after the internal sequencing of the test module has been completed successfully. Third, a subroutine is also produced for every operator message defined in NOPAL specification. Lastly, the OPAL code for properly invoking test-module routines and for inserting necessary control logics are generated. This last step is begun after the inter-test-module (i.e., external) sequencing of the specification has been successfully done.

The product of this phase is an OPAL test program in accordance with the NOPAL specification provided by the user for testing the UUT. This OPAL program may be augmented by a library of OPAL routines, if the user so desires and supplies it. Any routine (NOPAL function) which is used in the NOPAL specification and whose OPAL code is not supplied at this stage by the user is expected to be resolved later at OPAL compile time, or even at run time. The complete OPAL program is then ready for compilation and execution (actual testing of the UUT) in a given ATE system.

Chapter 7 describes Phase III in more detail.

CHAPTER 5

SYNTAX ANALYSIS AND ASSOCIATIVE MEMORY

5.1 INTRODUCTION

The first phase of the NOPAL Processor performs syntax and local semantic analysis of specification statements. At the end of the analysis, each NOPAL statement is encoded and stored in simulated associative memory for ease in further processing. As shown in Figure 5.1, the first phase consists of a Syntax Analysis Program (SAP). SAP itself is generated automatically by a meta-processor, Syntax Analysis Program Generation (SAPG), by inputting the formal specification of the NOPAL language in a meta-language, Extended Backus Normal Form with Subroutine Calls (EBNF/WSC).

Section 5.2 discusses the EBNF/WSC, SAPG, and SAP in more detail.

SAP incorporates six types of supporting routines which must be composed manually: Lexical Analyzer, Error Stacking, Recognizer, Encoding/Saving/Storing/, Semantics Checking and Housekeeping. These are covered in Section 5.3.

At the end of each NOPAL statement, a storing routine is invoked to store the statement in the simulated associative memory using a store/retrieve package. The store/retrieve package is presented in Section 5.4.

Section 5.5 describes the processing of two NOPAL specification reports and a syntax error report by SAP. A source specification report is produced by the lexical analyzer as a by-product. It is a listing of the NOPAL statements as entered by the user. Another reformatted specification report is generated by retrieving the statements

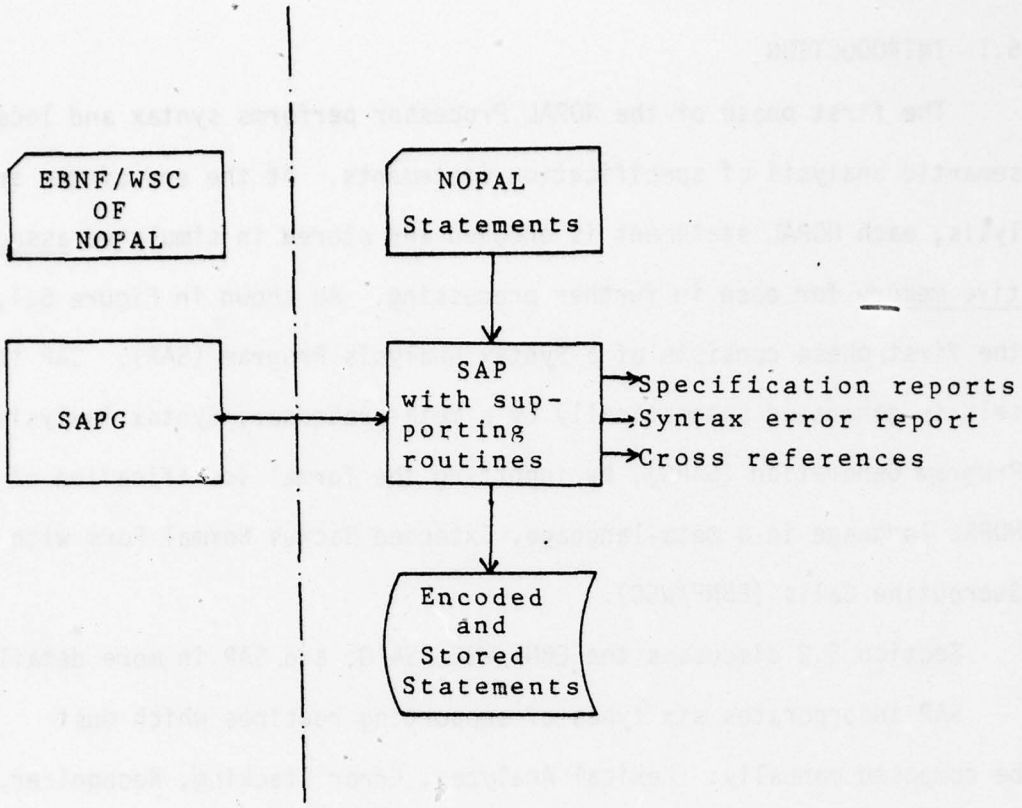


FIGURE 5.1

SAP WITH SAPG AND OUTPUTS

from the associative memory. This is a reformatted, reorganized NOPAL specification listing for easy readability. The syntax error report contains error and/or warning messages generated by SAP.

Finally, Section 5.6 presents the part of the system that generates a set of cross reference reports, which are summarized in Table 5.1.

5.2 EBNF/WSC, SAPG, and SAP.

The Syntax Analysis Program (SAP) for processing the NOPAL statements is generated by the Syntax Analysis Program Generator (SAPG). The input to SAPG is a specification of the NOPAL language in a meta-language Extended Backus Normal Form with Subroutine Calls (EBNF/WSC).

SAPG and EBNF/WSC were developed at the University of Pennsylvania by the Data Definition Language Project [RAM 73 and FRE 72]. A brief review of SAPG and EBNF/WSC is given in the following sections.

5.2.1 SPECIFICATION OF NOPAL USING EBNF/WSC and SAPG

The EBNF/WSC includes and extends the concepts of the conventional Backus Normal Form (BNF). The BNF itself is a language consisting of statements (production rules) which describe the syntax of formal languages (here NOPAL). BNF uses four meta-linguistic characters <,>, ::=, and | . Sequences of characters enclosed in angle-brackets are called non-terminals, and denote names of syntactic units and for which substitutions may be made. Sequences of characters not enclosed in angle-brackets are called terminals, which represent keywords or characters in NOPAL. Each production rule in the BNF is of the form "L ::= R". L is a non-terminal symbol and R is one or more alternative sequences of terminal or non-terminal symbols that can be substituted for L. The alternatives are separated by the meta-linguistic symbol "|".

TABLE 5.1 CROSS REFERENCE REPORTS

Report Number	Name of Report	Name of Entity Cross References	Name of Entities Cross Referenced with
1	XREF-ATTR	Variable, data	Source statement numbers, attributes
2	DIAG-TEST	Diagnosis	Test modules
3	MESS-DIAG-TEST	Operator message	Diagnoses, test modules
4	COMP-DIAG-TEST	Affected component	Diagnoses, test modules
5	UUT.PT-TEST-ATE.PT	UUT connecting point	Test modules, ATE connecting points
6	FUNC-TEST	ATE function	Test modules

To facilitate language description, BNF was extended to EBNF with two meta-linguistic symbols: [ ] representing optionality and [ ]* representing repetition of zero or more times. Because of unavailability of the square brackets [ ] in regular keypunches, the left bracket [is replaced by &-0 multi-punch immediately followed double quote " , and the right bracket] by double quote immediately followed by &-0 multi-punch. Note that in the computer printouts the &-0 multi-punch becomes blank, hence the two brackets appear as double quotes.

A description of the NOPAL language using EBNF, without subroutine calls, has been presented in Figure 3.1. The EBNF, like BNF, is sufficient to describe the syntax of the NOPAL language; it is not capable of describing the semantics of the language. Therefore, EBNF was expanded to allow subroutine names to be embedded within it. Hence the name "EBNF with Subroutine Calls" (EBNF/WSC) was adopted.

The EBNF/WSC specification of NOPAL constitutes the input to SAPG. It consists of the syntax specification as well as subroutine names enclosed in slashes "/". The embedded subroutine name indicates need to the respective subroutine upon successful recognition of the preceding syntactic unit. Thus, these subroutines, incorporated in SAP, enable checks of local semantics. They produce error messages, encode /save, and store away statements. The invocation of these subroutines is incorporated in the automatically generated SAP. The sub-routines themselves are written manually.

The specification of the NOPAL language in EBNF/WSC is shown in Figure 5.2. Unlike the human-oriented EBNF specification presented

```
EBNF/WSC
Line No.                                                                                    EBNF
                                                                                           REF. NO
 1    <NOPAL_SPECIFICATION> ::=  " <NOPAL_STMTS> /CLRERRF/ " *                             53
 2                                          /STMT_FL/ <NOPAL_SPECIFICATION>
 3    <IDENTIFIER>  ::=  /NAMEREC/                                                         18
 4    <INTEGER>  ::=  /INTEGER/                                                            16
 5    <UNSIGNED_INTEGER>  ::=  /POSINTR/                                                   13
 6    <NUMBER>  ::=  /NUMBER/                                                              9
 7    <UNSIGNED_NUMBER>  ::=  /POSNUMB/                                                    11
 8    <STRING_CONST>  ::=  /STRREC/                                                        1
 9    <CHAR_STRING>  ::=  /CHARSTR/                                                        2
10    <ARITH_EXPR>  ::=  /SETAREX/ "<SIGN>/AREXS1/"  <TERM>                                21
11                              "<ADD_OP>/AREXS1/ <TERM>" *
12    <TERM>  ::=  <FACTOR>  "<MULT_OP>/AREXS1/ <FACTOR>" *                                22
13    <SIGN>  ::=  <ADD_OP>                                                                10
14    <ADD_OP>  ::=  + | -                                                                 29
15    <MULT_OP>  ::=  * | /                                                                28
16    <EXPONENTIATION>  ::=  /EXPONET/                                                     23
17    <FACTOR>  ::=  <PRIMARY>  "<EXPONENTIATION>/AREXS1/ <PRIMARY>" *                     23
18    <PRIMARY>  ::=  /AXER1/ (<ARITH_EXPR>/AREXSAX/ )                                     24
19                  | <UNSIGNED_NUMBER>/AREXS1/
20                  | <FUNCTION_CALL>/AREXSVF/
21    <FUNCTION_CALL>  ::=  <FUNCTION_ID> /SETVF/  "(/VFS0/ <ARGUMENT>                     25
22                                          ",/VFS1/<ARGUMENT>" * /RPAR/)/VFS1/"
23    <FUNCTION_ID>  ::=  <IDENTIFIER>                                                     26
24    <ARGUMENT>  ::=  /ARGSUBS/<STRING_CONST>/VFS2/                                       27
25                   | <ARITH_EXPR>/VFSAX/
26    <VARIABLE>  ::=  <FUNCTION_CALL> /CKSTR/                                             17
27    <IF_CLAUSE>  ::=  IF <BOOLEAN_TERM> /IFCOND/ THEN                                    31
28    <BOOLEAN_TERM>  ::=  /SETBEXP/<BOOLEAN_FACTOR>  "<OR>/BEXPS1/<BOOLEAN_FACTOR>" *     32
29    <OR>  ::= /OR_OP/                                                                    32
30    <BOOLEAN_FACTOR>  ::=  <BOOLEAN_PRIMARY>  "&/BEXPS1/ <BOOLEAN_PRIMARY>" *            33
31    <BOOLEAN_PRIMARY>  ::=  /BXER2/( <BOOLEAN_TERM>/BEXPSBX/ )                           34
32                              | <ARITH_EXPR>/BEXPSAX/ /BXER1/<RELATION>/BEXPS1/
33                                                  <ARITH_EXPR>/BEXPSAX/
34    <CONN_DIM_EX>  ::= <CONNECTOR>  "<DIMENSION>/CDESDM/"                                37
35    <CONNECTOR>  ::=  < <CONNECTOR_ID>  ",<CONNECTOR_ID>" * /CDER2/ >                    38
36                   | <CONNECTOR_ID>
37    <CONNECTOR_ID>  ::=  /CDER1/<IDENTIFIER>/CDESID/                                     39
38    <DIMENSION>  ::=  /DIMREC/                                                           40
```

FIGURE 5.2 EBNF/WSC FOR NOPAL

```
<VAL_DIM_EX>  ::=  <ARITH_EXPR>/FDESAX/  "<DIMENSION>/FDESDM/"
<FUNC_DIM_EX>  ::=  /SETFDE/<FUNC_TERM>  "<ADD_OP>/FDES1/<FUNC_TERM>" *
<FUNC_TERM>  ::=  <FUNC_FACTOR>  "<MULT_OP>/FDES1/<FUNC_FACTOR>" *
<FUNC_FACTOR>  ::=  "<FUNC_MODIFIER>/FDESMOD/<MULT_OP>/FDES1/" <FUNC_PRIMARY>
<FUNC_PRIMARY>  ::=  /FDER1/( <FUNC_DIM_EX>/FDESFDE/ )
                   | <FUNCTION_ID>/FDES1/  "(/FDES1/<FUNC_ARG>
                                           ",/FDES1/<FUNC_ARG>" * )/FDES1/"
<FUNC_ARG>  ::=  * /FDES1/
               | <STRING_CONST> /FDES1/
               |  "<RELATION>/FDES1/"  <VAL_DIM_EX>  "<PLUS_MINUS>/FDES1/
                                                     <VAL_DIM_EX>"
<FUNC_MODIFIER>  ::=  <UNSIGNED_NUMBER>
<NOPAL_STMTS>  ::=  /SPECERR/ "NOPAL"  <SPECIFICATION> "<SPEC_NAME>/SVLBL/"
                                          /STSPEC/ /STMTEND/
                      | <TEST_MODULE_SPEC>
                      | <UUT_SPEC>
                      | <ATE_SPEC>
                      | END  "<SPEC_NAME>/SVLBL/"  /STEND/ /STMTEND/
<SPECIFICATION>  ::=  /SPECIF/
<SPEC_NAME>  ::=  <LABEL>
<LABEL>  ::=  /LABEL/
<TEST_MODULE_SPEC>  ::=  <TEST_STEP>
                       | <DIAGNOSIS_DEFINITION>  "<DIAGNOSIS_DEFINITION>" *
                       | <MESSAGE_DEFINITION>  "<MESSAGE_DEFINITION>" *
<TEST_STEP>  ::=  TEST  "<TEST_LABEL>/SVLBL/"  /STTEST/ /STMTEND/
                | <STIMULI> <WAVEFORM_ID>/STSTIM/  ":<BACK_REF>"  /STMTEND/
                | <MEASUREMENT> <WAVEFORM_ID>/STMEAS/  ":<BACK_REF>"  /STMTEND/
                | LOGIC <WAVEFORM_ID>  ":"  <LOGIC_DIAG_LIST>/STLOG/ /STMTEND/
                | <WAVEFORMS>  "<WAVEFORMS>" *
<TEST_LABEL>  ::=  <LABEL>
<STIMULI>  ::=  /STIMULI/
<WAVEFORM_ID>  ::=  "<LABEL>/SVLBL/"  "(/TSMER1/<LABEL>/SVLBL2/)"
<BACK_REF>  ::=  /CONJ1/<BACK_REFERENCE>  "<DECLARATION>" * /STWAVFM/
<MEASUREMENT>  ::=  /MEASURE/
<LOGIC_DIAG_LIST>  ::=  /LOGID/<LOGOP_DIAGLBL>  ",<LOGOP_DIAGLBL>" *
<LOGOP_DIAGLBL>  ::=  /LOGIC2/<LOGICAL_OPERATOR> <DIAG_LABEL>/SLOPLBL/
<LOGICAL_OPERATOR>  ::=  /LOGICOP/
<WAVEFORMS>  ::=  <CONJUNCTION> | <ASSERTION>  "<ASSERTION>" *
```

FIGURE 5.2 EBNF/WSC FOR NOPAL (continued)

```
77    <CONJUNCTION>  ::=  <CONJUNCT><WAVEFORM_ID> /COLON/: /CONJ1/ <CONJUNCTION_BODY>     69
78                                    "<DECLARATION>" * /STWAVFM/ /STMTEND/
79    <CONJUNCT>  ::=  /CONJUNC/                                                         69
80    <CONJUNCTION_BODY>  ::=  <BACK_REFERENCE> | <TRIPLET_CONJUNCT>                     72
81    <TRIPLET_CONJUNCT>  ::=  <IF_CONJUNCTION> | <SIMPLE_CONJUNCTION>                   73
82    <SIMPLE_CONJUNCTION>  ::=  <TRIPLET>  "& <TRIPLET>" * /STRIPT/                     74
83    <TRIPLET>  ::=  (<CONN_DIM_EX>/CONJ5/ <RELATION>/CJSREL/ <FUNC_DIM_EX>/CJSFDE/)    75
84                  | <CONN_DIM_EX>/CONJ5/ <RELATION>/CJSREL/ <FUNC_DIM_EX>/CJSFDE/
85    <RELATION>  ::=  /RELREC/                                                          36
86    <IF_CONJUNCTION>  ::=  <IF_CLAUSE> <SIMPLE_CONJUNCTION>                            76
87                                    "ELSE <TRIPLET_CONJUNCT>"
88    <BACK_REFERENCE>  ::=  "SAME"  AS/SAME1/ <STIM_MEAS_LABEL> /BRSLBL/                77
89                              "<EXCEPT> <SIMPLE_CONJUNCTION>"
90    <STIM_MEAS_LABEL>  ::=  <LABEL>                                                    71
91    <EXCEPT>  ::=  /EXCEPT/                                                            77
92    <DECLARATION>  ::=  <VARIABLE_TYPE>  ":"  <VARIABLE_LIST>                          78
93    <VARIABLE_TYPE>  ::=  /SRC_TGT/                                                    79
94    <VARIABLE_LIST>  ::=  (<VAR_ELEM>  ",<VAR_ELEM>" * /RPAR/)                        80
95                        |  <VAR_ELEM>  ",<VAR_ELEM>" *
96    <VAR_ELEM>  ::=  /DCLER1/<IDENTIFIER>/DCLSID/ <SUBSCRIPT_LIST>                     80
97    <SUBSCRIPT_LIST>  ::=  "(<SUBSCRIPT>  ",<SUBSCRIPT>" * /RPAR/)"  /DCLSVAR/         80
98    <SUBSCRIPT>  ::=  /ARGSUBS/<ARITH_EXPR> /DCLSUBS/                                  80
99    <ASSERTION>  ::=  <ASSERT> <WAVEFORM_ID> /ASRT1/: <ASSERTION_BODY>                 81
100                                   "<DECLARATION>" * /STWAVFM/ /STMTEND/
101   <ASSERT>  ::=  /ASSERT/                                                            81
102   <ASSERTION_BODY>  ::=  <IF_ASSERTION> | <SIMPLE_ASSERTION>                         82
103   <SIMPLE_ASSERTION>  ::=  /GETASRT/<RELATIONAL_EXPR>  "<PLUS_MINUS><ARITH_EXPR>     83
104                                                  /ASRANGE/  "%/ASPC/"  "
105   <RELATIONAL_EXPR>  ::=  <ARITH_EXPR>/ASEXPS/ /ASRT3/ <RELATION>/ASREL/             35
106                                 <ARITH_EXPR> /ASEXPS/
107   <PLUS_MINUS>  ::=  /PLUSMIN/                                                       83
108   <IF_ASSERTION>  ::= <IF_CLAUSE> <SIMPLE_ASSERTION>  "ELSE <ASSERTION_BODY>"        84
109   <DIAGNOSIS_DEFINITION>  ::=  <DIAGNOSIS>/DIAG1/ <DIAG_LABEL>/SVLBL/  ":"           85
110                                       <DIAG_BODY> /STDIAG/ /STMTEND/
111   <DIAGNOSIS>  ::=  /DIAGNOS/                                                        85
112   <DIAG_LABEL>  ::=  <LABEL>                                                         86
113   <DIAG_BODY>  ::=  <KEYWORD_DIAG> | <POSITIONAL_DIAG>                               87
114   <POSITIONAL_DIAG>  ::=  "<OPERATOR_MESSAGE>"  ",<OPERATOR_RESPONSE>"               88
```

FIGURE 5.2 EBNF/WSC FOR NOPAL (continued)

```
115     <OPERATOR_MESSAGE>  ::=  ( "<AFFECTED_COMPONENTS>"    ", "<OTHER_PARAMETERS>          89
116                             /SPARM/"    ", "<TYPE>/STYP/"    ",<TIMING>"  "  " /RPAR/)
117                           | <AFFECTED_COMPONENTS>
118     <AFFECTED_COMPONENTS>  ::=  <COMPONENT_ELEM>  "<AND_OR>/CMPSLOP/                 90
119                                                    <COMPONENT_ELEM>" * /CMPRLOP/
120     <COMPONENT_ELEM>  ::=  <IDENTIFIER>/CMPSID/  "(/CMPSFF/<COMPONENT>/CMPSID/       91
121                              "<AND_OR>/CMPSLOP/<COMPONENT>/CMPSID/" * /FFRSET/)"      &
122                          | <COMP_FAIL_SEQN>/SVCMPFL/                                  93
123     <COMPONENT>  ::=  <IDENTIFIER>                                                   92
124     <AND_OR>  ::=  /ANDOROP/                                                         90
125     <OTHER_PARAMETERS>  ::=  (<MSG_ARG_REQD>  ",<MSG_ARG_REQD>" * /RPAR/)            94
126                            | <MSG_ARGUMENT>
127     <MSG_ARG_REQD>  ::=  /OPMSG3/ <MSG_ARGUMENT>                                     94
128     <MSG_ARGUMENT>  ::=  <STRING_CONST>/MASTR/ | <NUMBER>/MANUM/                     95
129                        | <IDENTIFIER>/MAVAR/ <SUBSCRIPT_LIST>
130     <TYPE>  ::=  <MESSAGE_LABEL>                                                     96
131     <TIMING>  ::=  <NUMBER>/TIME1/  "<TIME_DIMENSION>/TIME2/"                        97
132     <TIME_DIMENSION>  ::=  /TIMDIM/                                                  43
133     <OPERATOR_RESPONSE>  ::=  ?/OPRPS1/                                              98
134                             | (<OP_VAR_LIST>/RPAR/)  ","   "?/OPRPS1/"
135                             | <OP_VAR_LIST>  "?/OPRPS1/"
136     <OP_VAR_LIST>  ::=  <OP_VAR>  ",<OP_VAR>" *                                      99
137     <OP_VAR>  ::=  /OPRPS3/<IDENTIFIER>/OPRPS2/ <SUBSCRIPT_LIST>                     99
138     <KEYWORD_DIAG>  ::=   "OPERATOR /DIAGER2/<MESSAGE>:"  <DIAG_KEYWD>              100
139                                                              ",<DIAG_KEYWD>" *
140     <DIAG_KEYWD>  ::=   "AFFECTED"  <COMP_FAIL> /KEYEQ/= <AFFECTED_COMPONENTS>      101
141                       | "OTHER"  <PARAMETER> /KEYEQ/= <OTHER_PARAMETERS>/SPARM/
142                       | TYPE/KEYEQ/=/MSGER1/<TYPE>/STYP/ | TIME /KEYEQ/= <TIMING>
143                       | <RESPONSE> /KEYEQ/= <OPERATOR_RESPONSE>
144     <PARAMETER>  ::=  /PARAMET/                                                      101
145     <RESPONSE>  ::=  /RESPONS/                                                       101
146     <MESSAGE_DEFINITION>  ::=  <MESSAGE>/MSGER1/<MESSAGE_LABEL>/SVLBL/  ":"          102
147                                  "ALIAS /MSGER2/ = <SYNONYM>/SVSYN/,"
148                                    "TEXT/KEYEQ/="  <MESSAGE_TEXT>/STMSG/ /STMTEND/
149     <MESSAGE>  ::=  /MESSAGE/                                                        102
150     <MESSAGE_LABEL>  ::=  <LABEL>                                                    103
151     <SYNONYM>  ::=  <IDENTIFIER>                                                     104
152     <MESSAGE_TEXT>  ::=  /TXTBAD/<TEXT_ELEM>  "  "," <TEXT_ELEM>" *                 105
153     <TEXT_ELEM>  ::=  <CHAR_STRING>/TXTCH/                                           106
```

FIGURE 5.2 EBNF/WSC FOR NOPAL (continued)

```
154   <UUT_SPEC>  ::=  <UUT_COMPONENT_FAILURE>  "<UUT_COMPONENT_FAILURE>" *                    107
155                  | <UUT_CONNECTION_POINT>  "<UUT_CONNECTION_POINT>" *
156   <UUT_COMPONENT_FAILURE>  ::= <COMP_FAIL> "<COMP_FAIL_SEQ#>/SVL8L/"  "!" /TBLER2/       108
157                    <COMPONENT>/UUTCMPF/ ",<COMP_FAIL_KEYWD>" */STCOMP//STMTEND/
158   <COMP_FAIL_SEQ#>  ::=  <ENTRY_SEQ#>                                                     109
159   <ENTRY_SEQ#>  ::=  <UNSIGNED_INTEGER>                                                   110
160   <COMP_FAIL>  ::=  /COMPFL/                                                              108
161   <COMP_FAIL_KEYWD>  ::=  /TBLER1/ALIAS = <SYNONYM>/SVSYN/                                111
162                           | <FAILURE> "<FUNCTION>" =<FAILURE_FUNCTION>/FLSFF/
163                           | <PARAMETER> = <PARM_LIST>/SVPARML/
164                           | INDEX = <FAILURE_INDEX>/FLSIDX/
165                           | <PROTECT> = <PROTECTION>
166                           | <COMMENTS>
167   <FAILURE>  ::=  /FAILURE/                                                               111
168   <FUNCTION>  ::=  /FUNCION/                                                              111
169   <FAILURE_FUNCTION>  ::=  <FUNCTION_ID>                                                  112
170   <PARM_LIST>  ::=  (<PARM_NAME>  ",<PARM_NAME>" * /RPAR/)                                113
171                   | <PARM-NAME>
172   <PARM_NAME>  ::=  /TBLER2/<IDENTIFIER> /PMSID/                                          114
173   <FAILURE_INDEX>  ::=  <INTEGER>                                                         115
174   <PROTECT>  ::=  /PROTECT/                                                               111
175   <PROTECTION>  ::= (<COMP_LABEL>  ",<COMP_LABEL>" * /RPAR/)                              116
176                   | <COMP_LABEL>
177   <COMP_LABEL>  ::=  <COMPONENT_ELEM>                                                     117
178   <COMMENTS>  ::=  "<COMMENT> /KEYEQ/="  <CHAR_STRING>/SCOMT/                             118
179   <COMMENT>  ::=  /COMMENT/                                                               111
180   <UUT_CONNECTION_POINT>  ::=  <UUT_POINT> "<ENTRY_SEQ#>/SVSEQ#/"  "!" /TBLER2/          119
181                    <UUT_POINT_ID>/UUTPNT/ ",<UUT_POINT_KEYWD>" */STUUTPT//STMTEND/
182   <UUT_POINT>  ::=  /UUT_PNT/                                                             119
183   <UUT_POINT_ID>  ::=  <IDENTIFIER>                                                       120
184   <UUT_POINT_KEYWD>  ::=  /TBLER1/ALIAS = <SYNONYM>/SVSYN/                                121
185                           | <CONNECT> = <UUT_CONNECTOR>
186                           | LIMIT = <PROTECTIVE_LIMITS>
187                           | <COMMENTS>
188   <CONNECT>  ::=  /CONNECT/                                                               121
```

FIGURE 5.2 EBNF/WSC FOR NOPAL (continued)

```
<UUT_CONNECTOR>  ::=  (/TBLER2/<CONN_TYPE>/SVCNTYP/                                    122
                                        ",/TBLER2/<CONN_POINT>/SVCNPT/"  /RPAR/)
                                | <CONN_TYPE> /SVCNTYP/
<CONN_TYPE>  ::=  <IDENTIFIER>                                                        123
<CONN_POINT>  ::=  <IDENTIFIER>                                                       124
<PROTECTIVE_LIMITS>  ::=  /GETLMT/( "<DIMENSION>/PMSDM/"   ", "<MAX_LIMIT>           125
                                   /PMSHL/"   ", "<MIN_LIMIT>/PMSLL/"
                                ",/TBLER2/<REFERENCE_POINT>/PMSRPT/"  "  " /RPAR/)
                                   | <DIMENSION>/PMSDM/
<MAX_LIMIT>  ::=  <NUMBER>                                                            126
<MIN_LIMIT>  ::=  <NUMBER>                                                            127
<REFERENCE_POINT>  ::=  <UUT_POINT_ID>                                                128
<ATE_SPEC>  ::=  <ATE_FUNCTION>  "<ATE_FUNCTION>" *                                   129
              | <ATE_CONNECTION_POINT>  "<ATE_CONNECTION_POINT>" *
<ATE_FUNCTION>  ::=  <FUNCTION> "<ENTRY_SEQ#>/SVSEQ#/"   ":"  /TBLER2/               130
                  <FUNCTION_ID>/ATEFUNC/ ",<FUNCTION_KEYWD>" */STFUNC//STMTEND/
<FUNCTION_KEYWD>  ::=  /TBLER1/ALIAS = <SYNONYM>/SVSYN/                               131
                           | "<FUNCTION>"  TYPE = <FUNCTION_TYPE>/FNSTYP/
                           | #PINS = <UNSIGNED_INTEGER>/FN#PINS/
                           | <PARAMETER> = <PARM>
                           | VALUE  "RETURNED"  = <VALUES_RETURNED>/FNSVAL/
                           | <COOPERATION> = <COOP_FUNCTIONS>
                           | <COMMENTS>
<FUNCTION_TYPE>  ::=  /FNER1/ S | M | F | E | C                                       132
<PARM>  ::=  (<PARM_NAME>  ", "<PARM_TYPE>/PMSTYP/"                                   133
                           ", "LIMIT/KEYEQ/=" <PROTECTIVE_LIMITS>"  "  /RPAR/)       135
           | <PARM_NAME>
<PARM_TYPE>  ::=  S | T                                                               134
<VALUES_RETURNED>  ::=  <CHAR_STRING>                                                 136
<COOPERATION>  ::=  /COOPERA/                                                         131
<COOP_FUNCTIONS>  ::=  (/TBLER2/<FUNCTION_ID>/FNSCF/                                  137
                                        ",/TBLER2/<FUNCTION_ID>/FNSCF/" * /RPAR/)
                     | /TBLER2/<FUNCTION_ID> /FNSCF/
```

FIGURE 5.2 EBNF/WSC FOR NOPAL (continued)

```
222     <ATE_CONNECTION_POINT>  ::=  <ATE_POINT> "<ENTRY_SEQN>/SVSEQN/"    ":"  /TBLER2/        138
223                         <ATE_POINT_ID>/SVLBL/ ",<ATE_POINT_KEYWD>" */STATEPT//STMTEND/
224     <ATE_POINT>  ::=  /ATE_PNT/                                                              138
225     <ATE_POINT_ID>  ::=  <IDENTIFIER>                                                        139
226     <ATE_POINT_KEYWD>  ::=  /TBLER1/ALIAS = <SYNONYM>/SVSYN/                                 140
227                                 | <UUT_POINT> = <UUT_POINTS>
228                                 | <COMMENTS>
229     <UUT_POINTS>  ::=  (<UUT_POINT_ID>/SVPTID/  ",<UUT_POINT_ID>/SVPTID/" * /RPAR/)          141
230                        | /TBLER2/<UUT_POINT_ID>/SVPTID/
```


FIGURE 5.2 EBNF/WSC FOR NOPAL (continued)

in Figure 3.1 the EBNF/WSC specification includes the following two modifications:

(1) the EBNF specification has been restructured to conform to restrictions explained in Section 5.2.3, imposed by the SAPG processor.

(2) the names of the invoked subroutines are embedded in EBNF (enclosed in slashes).

The left-hand column in Figure 5.2, marked "EBNF/WSC Line Number", shows the line (or card) number of EBNF/WSC for NOPAL. The right-hand column, marked "EBNF Reference Number', indicates the statement (production) number of the corresponding EBNF statement of Figure 3.1. Where one EBNF statement corresponds to more than one EBNF/WSC statement, the same EBNF reference number appears in all of the EBNF/WSC statements.

To illustrate the relationship between EBFF and EBNF/WSC statements, the following is an example from the EBNF specification of Figure 3.1 (Statement 85).

```
<DIAGNOSIS_DEFINITION>   ::=  DIAGNOSIS <DIAG_LABEL> [:]
                                          <DIAG_BODY> ;
```

In the EBNF/WSC specification on lines 109-110 of the Figure 5.2, the above becomes the following:

```
<DIAGNOSIS_DEFINITION> ::= <DIAGNOSIS> /DIAGI/ <DIAG_LABEL>
                                  /SVLBL/ [:] <DIAG_BODY>
                                  /STDIAG/ /STMTEND/
```

This means that the non-terminal <DIAGNOSIS_DEFINITION> starts with the syntactic unit <DIAGNOSIS> . The corresponding recognizer

routine will recognize the keyword DIAGNOSIS. If this keyword is successfully recognized, the subroutine DIAG1 is called. This routine will allocate and stack an error message code for a missing succeeding syntactic unit (which in this case is <DIAG_LABEL>). Then another procedure is called to recognize the next syntactic unit<DIAG_LABEL> . If <DIAG_LABEL> is successfully recognized, the routine SVLBL is invoked, which will encode and save the recognized token. Otherwise, the error message will be sent to the user. Then a colon (:) may optionally follow. Next comes the last non-terminal <DIAG_BODY> (it will define the ATE operator message and operator response in another production). If the foregoing is successful, the subroutine STDIAG is called to store the statement in the simulated memory, by calling, in turn, the STORE subsystem (to be explained later). Finally, the subroutine STMTEND is invoked. It will check the statement end marker (;) and increment the statement number.

Further examples of inserting subroutine calls into the EBNF/WSC will be given later when each category of subroutines (such as recognizer and saving/encoding) is discussed in the following sections.

In summary, SAP is generated by SAPG based on the EBNF/WSC specification of NOPAL and linked with the subroutines. Then SAP accepts NOPAL statements and checks them for syntactic correctness and some local semantics, encodes, and stores the statements in the simulated associative memory for further processing.

5.2.2 HOW SAPG PRODUCES SAP

As indicated in Figure 5.1, SAPG produces SAP based on the EBNF/WSC specification of the NOPAL language. The design of SAPG is

documented in [RAM 73] and its implementation in [FRE 72]. The operation of SAPG is briefly summarized in the following paragraphs, but the above documentations should be referred to for more details.

SAPG itself is a small compiler which accepts as input a formal description of a given language L (here NOPAL) expressed in a meta-language EBNF/WSC. It produces a PL/1 program (SAP) which analyzes the statements in the language L and coordinates the encoding and storing of the statements in an internal form. SAPG processes the set of EBNF/WSC source statements (i.e., productions) in the following three passes.

In Pass 1, it performs lexical analysis of the EBNF/WSC productions and encodes them in an "Encoded Table." Non-Terminals appearing on the left-hand side of the symbol ::= in a production are placed in a "Symbol Table," while non-terminals appearing on the right-hand side are put into a "Work Table." Subroutine calls and terminal symbols are placed in subroutine and terminal symbol tables respectively. Altogether SAPG maintains five internal tables(Encoded, Symbol, Work, Terminal, and Subroutine).

In Pass 2, SAPG scans the Encode Table to resolve the symbolic references in the Work Table (i.e., finds non-terminals on the right-hand side of the original production). It checks that each non-terminal on the right-hand side of a production is defined, and links it to the corresponding entry in the non-terminal symbol table. Undefined, as well as circularly defined, nonterminals are detected in this phase and reported as errors.

In Pass 3, SAPG generates SAP code in PL/I. This phase of SAPG is entered only if no errors were detected in the first two passes. For each EBNF/WSC production, a PL/I Procedure is generated, which returns a 1-bit value. The procedure returns a 0 value on failure of recognition of the first syntactic unit on the right-hand side of the symbol ::= in the production. Otherwise, it returns a 1 value. The exclusive nature of EBNF production rules and alternatives is implemented by PL/I IF-THEN-ELSE statements. Repetition brackets ([...]*) in a production cause generation of GOTO statement in a place of SAP to scan again the first syntactic unit of the group. Each subroutine name embedded in slashes in EBNF/WSC becomes a "call" statement for the subroutine. Calls to the lexical scanner LEX and other "housekeeping" subroutines are also inserted in SAP, as indicated.

As an example of the SAP code that SAPG produces, consider the following representative production rules (EBNF/WSC lines 109-112 and 59 of Figure 5.2):

```
< DIAGNOSIS_DEFINITION > ::= < DIAGNOSIS > /DIAG1/
                             < DIAG_LABEL>/SVLBL/ [:]
                             < DIAG_BODY >/STDIAG/
                               /STMTEND/
```

```
<DIAGNOSIS >   ::= /DIAGNOS/
<DIAG_LABEL >  ::=   <LABEL >
<LABEL  >   ::=  /LABEL/
```

The corresponding PL/I code generated for it by SAPG in the third pass is:

```
DIAGNOSIS_DEFINITION: PROCEDURE RETURNS (BIT(1));
   CALL $MARK;
   IF DIAGNOS THEN
      DO; CALL $POPF;
          CALL DIAG1;
          IF LABEL THEN
             DO; CALL $POPF;
                 CALL SVLBL;
                 $SYS_049: CALL LEX;
                 IF LEXBUFF = ':' THEN
                    DO; CALL LEXENAB; END;
                 ELSE;
          IF DIAG_BODY THEN
             DO; IF ERRORSW THEN
                 DO; CALL $SUCCES;
                     RETURN('1'B);
                 END;
             ELSE;
             CALL STDIAG;
             CALL STMTEND;
```

```
                CALL $SUCCES;
                RETURN('1'B);
              END;
            END;
          ELSE DO; CALL $FAIL; RETURN('1'B); END;
        END;
      ELSE DO; CALL $FAIL; RETURN('0'B); END;
    END DIAGNOSIS_DEFINITION;
```

The above code would become an internal procedure in SAP. The subroutines beginning with a dollar sign ($) are "housekeeping" routines, which are internal to the mechanisms of SAPG. Normally, they do not concern the language definer. These routines are further discussed in Section 5.3.6. Two "recognizer" routines (DIAGNOS and LABEL) are illustrated in the above example. If a subroutine appears alone as the right part (i.e., the part to the right of the symbol ::=) of a production, the subroutine is determined by SAPG as a recognizer routine. For example, DIAGNOS is a recognizer routine because of the production "<DIAGNOSIS > ::= /DIAGNOS/ ". Recognizer routines and their references are further explained in Section 5.3.3. How SAPG generates the above PL/I code is briefly explained in the next paragraph.

Before generating the code for the production < DIAGNOSIS_DEFINITION >, SAPG has determined that

DIAGNOS and LABEL are recognizer routines and hence that <DIAGNOSIS > and <DIAG_LABEL > in this production are the non-terminals associated with the two recognizer routines. First, SAPG generates DIAGNOSIS_DEFINITION procedure header based on the production named <DIAGNOSIS_DEFINITION > . Then "CALL $MARK;" is generated to mark in the error stack the beginning of error codes for this production. Next comes the non-terminal <DIAGNOSIS > which has been determined to be associated with the recognizer routine DIAGNOS, hence "IF DIAGNOS THEN DO; CALL $POPF;" is produced. "CALL $POPF;" is generated to pop the top error code from the error stack, if any. Then, /DIAG1/ subroutine call is encountered; therefore, the corresponding "CALL DIAG1;" PL/I statement is generated. Then comes <DIAG_LABEL > , associated with the recognizer routine LABEL, as indicated, hence "IF LABEL THEN DO; CALL $POPF;" is produced. "CALL SVLBL:" is then generated due to the immediately following subroutine call /SVLBL/. Now a left bracket ([), signalling the beginning of an optionality group, is encountered, hence a unique PL/I statement label (in this case, "SYS_049:") is generated. Then comes the terminal symbol colon (:). A call to the lexical analyzer (LEX) is generated. If the current token (in LEXBUFF) is a colon,

a call (LEXENAB) to "enable" the LEX is also generated. Then < DIAG_BODY > follows, which is defined in another production. Thus, "IF DIAG_BODY THEN DO; IF ERRORSW THEN DO; CALL $SUCCES; RETURN('1'B); END; ELSE;" is produced. This causes SAP to restore the error stack (by calling $SUCCES) and return value 1 (true) if some errors have been detected (and hence error switch "ERRORSW" has been set) in the procedure for the production< DIAG_BODY > . Finally, two subroutine calls /STDIAG/ and /STMTEND/ follow, hence "CALL STDIAG;" and "CALL STMTEND;" are generated respectively. The PL/I DO group is wrapped up after restoring the error stack (CALL $SUCCES) and returning true. All ELSE groups except the very last one are completed by "CALL $FAIL; RETURN('1'B);", which issues an error message, restores the error-stack, and returns a true value. The last else group is closed in the same way except it returns a false value. At the end of the production <DIAGNOSIS_DEFINITION > , the "END DIAGNOSIS_DEFINITION;" is generated to end the procedure definition.

5.2.3. LIMITATIONS AND IMPLEMENTATION RESTRICTIONS OF SAPG

SAPG together with EBNF/WSC has proved to be a very useful tool for defining the NOPAL language, because it allows changes to the language to be made relatively easily during its development. The alternative of writing

the syntax and statement analysis program manually would be much more tedious. Although the SAPG approach has been found adequate for generating SAP for NOPAL, some limitations are mentioned below.

The first limitation is that SAPG only generates a SAP which performs statement-by-statement analysis to verify syntactic and local semantic correctness, the former directly and the later through subroutine calls. Consequently, global, inter-statement analysis is handled beyond the scope of SAP. Fortunately, NOPAL language is non-procedural, and each statement is independent. It turns out that the statement-by-statement local analysis is appropriate and adequate as a first pass. Global analysis of the NOPAL specification is one of the major tasks of the NOPAL processor to be discussed in Chapter 6.

SAPG has however several disadvantages. It is necessary for a language definer to define in PL/I all error-message routines and to insert the names of these routines in the EBNF/WSC specification. This is a tedious and time-consuming task, which requires a modification of the SAPG system. The SAPG system, in principle, could be designed and implemented in such a way that it would automatically generate the error messages for missing or incorrect syntactic units, based on the EBNF/WSC specification.

A standard facility (store/retrieve subsystem) for storing source language statements was developed and added to the original SAPG by the University of Pennsylvania MODEL project [RIN 76]. However, the store/retrieve procedures have been completely re-written to fulfill some special requirements of NOPAL, and to increase the efficiency of storing and retrieving. Note that routines for encoding of syntactic units temporarily saving of data and invoking the STORE sub-system (to store statements) must still be written manually.

There are some restrictions on the way EBNF/WSC is used to specify a language, due to the way SAPG has been implemented. SAPG does not generate a run time stack for syntactic units during syntax analysis and hence does not have a backtracking capability. Thus, the generated parser SAP is strictly sequential. As a consequence, the first restriction is that no production rule in the EBNF (and hence EBNF/WSC) specification can involve <u>left recursion</u>. A production is left recursive if the first symbol on the right-hand side of the symbol ::= is a non-terminal which is the left-hand side non-terminal itself, or which eventually references

the left-hand side non-terminal through valid substitution. A solution to this problem presented by the original SAPG system is to circumvent the left-recursion restriction by using the repetition feature of EBNF. In fact, left-recursion can easily be eliminated from a context free grammar [AHO 72]. A discussion of a method by which most recursion can be transformed into iteration may be found in [CAR 69].

A second restriction is that an optionality group must be distinguished by its first syntactic unit (terminal or non-terminal). In other words, the first syntactic unit which immediately follows the optionality group must be different from the first syntactic unit of the optionality group. For example, if "[,KEYWORD1 = ...] ,KEYWORD2 = ..." appeared in a production and the comma (",") itself were a lexical unit (in NOPAL processor the comma is truly a lexical unit), it would be impossible to determine by scanning a comma to which group the comma should belong. This would be overcome if the lexical routine were made to have back-tracking capability. Another solution is to treat the comma (",") as part of the keyword. The last alternative is to rewrite the EBNF specification to remove such occurrences, if possible.

The final restriction, also stemming from the strict

sequentiality of SAP, is that every alternative of a given production rule must be distinguishable by its first element. To illustrate this, consider the following example of four productions (statements 91, 92, 112, and 26 of EBNF for NOPAL in Figure 3.1):

< COMPONENT_CONJUNCT >::= < COMPONENT >
| <FAILURE_FUNCTION >(< COMPONENT >
[& < COMPONENT >]*)

< COMPONENT >::= < IDENTIFIER >

< FAILURE_FUNCTION >::= < FUNCTION_ID >

< FUNCTION_ID >::= < IDENTIFIER >

In the above example, both < COMPONENT > and < FAILURE_FUNCTION > are defined as < IDENTIFIER >. In order to recognize a string, say, "OPEN (RESISTOR)" as a < FAILURE_FUNCTION > (< COMPONENT >) and, in turn, as a < COMPONENT_CONJUNCT > a syntax parser with backtracking capability could first try the < COMPONENT > alone as first alternative, where "OPEN" would match but the remaining string "(RESISTOR)" would not. Then the parser would have to backtrack and try the second alternative, where "OPEN (RESISTOR)" would be found to match < FAILURE_FUNCTION > (< COMPONENT >) perfectly. Due to lack of such a backtracking capability, SAPG requires that each alternative to be taken must always

be determinable by the current token. To conform to this SAPG restriction, the above example could be rewritten by "factoring out" the common prefix < IDENTIFIER >as follows:

< COMPONENT_CONJUNCT >;:= < IDENTIFIER > < FUNC_OR_COMP >

< FUNC_OR_COMP > ::= [(< COMPONENT >[& < COMPONENT >]*)]

< COMPONENT > ::= < IDENTIFIER >

This restriction was the reason for restructuring some productions in the EBNF/WSC of NOPAL from the EBNF.

The above-mentioned two restrictions make the writing of the grammar somewhat awkward. EBNF/WSC for NOPAL has been written in this form by factoring out common syntactic units to higher levels in the parse tree and using keywords to uniquely identify paths or optionality groups. SAPG with these restrictions, however, is still adequate for the class of languages such as NOPAL.

In conclusion, while the SAPG system has some minor limitations and restrictions, it has been an adequate tool for defining NOPAL.

5.3 SUPPORTING SUBROUTINES FOR EBNF/WSC OF NOPAL

A flowchart showing SAPG and SAP with the types of supporting subroutines is shown in Figure 5.3. The

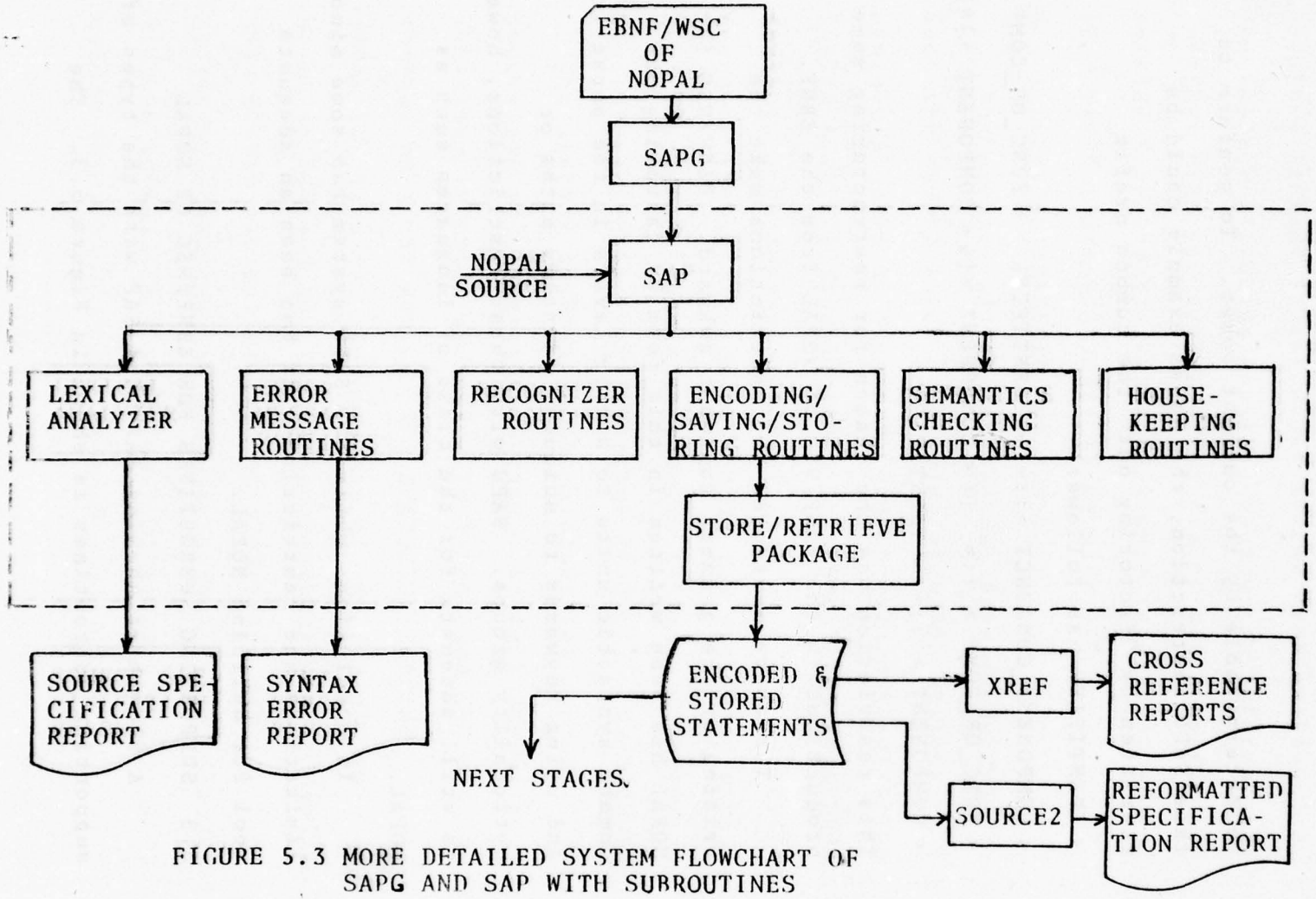


FIGURE 5.3 MORE DETAILED SYSTEM FLOWCHART OF SAPG AND SAP WITH SUBROUTINES

manually-written supporting routines are of the following six types:

1) lexical analyzer: scans the NOPAL input string and returns tokens of syntactic units to SAP or the recognizer routine for analysis;
2) error message stacking: composes and stacks error message codes;
3) recognizer: recognizes a class of input tokens, such as names and integers and returns true/false results to the SAP or another recognizer depending upon whether the recognition is successful or not;
4) encoding/saving/storing: compacts, temporarily saves, and stores NOPAL statements.
5) semantics checking: checks some local semantics of statements; and
6) housekeeping: required by the SAPG system in order to perform some services.

The above six types of routines are described in detail in the following subsections. At the end of each NOPAL statement a storing routine, which calls in turn the STORE subsystem of a STORE/RETRIEVE package, is invoked to store information of the statement. The STORE/

RETRIEVE package is discussed in the next Section 5.4. The two specification reports, source and reformatted, together with the syntax error report are presented in Section 5.5; the six cross reference reports as indicated in Table 5.1 are discussed in Section 5.6.

5.3.1 LEXICAL ANALYZER FOR NOPAL

The purpose of the lexical analyzer (scanner) is to scan the source input consisting of NOPAL statements for syntactic units or "tokens", and to return them to the Syntax Analysis Program (SAP) or the calling recognizer routine. The lexical analyzer (LEX or SCAN) is invoked whenever the next token is needed for syntactic checking.

The lexical analyzer routine is based upon the concept of finite state machines [CON 63]. Each state of the machine corresponds to a condition in the lexical processing of a character string. At each state, a character is read, an action is taken, and the machine changes to a new state. The character classes for the NOPAL language for the purposes of lexical analysis are shown in Table 5.2. The whole character set is divided into eleven categories such as alphabets, digits, delimiters, and special operators. The state transition diagram appears in Figure 5.4. Names enclosed by circles denote the states of the machine; two concentric circles

TABLE 5.2 CHARACTER CLASSES FOR NOPAL LANGUAGE

CLASS	CHARACTER SET	EXPLANATION
0	OTHERS	
1	(BLANK)	BLANK
2	A,B,..Y,Z,_,#,@,$	ALPHABETS AND _,#,@,$
3	0, 1, .. 9	DIGITS
4	'	QUOTE
5	<,>	BRACKETS
6	*	STAR
7	¬	NEGATION
8	/	SLASH
9	+	PLUS
10	|, &, ?	LOGICAL OPERATORS
11	-	MINUS

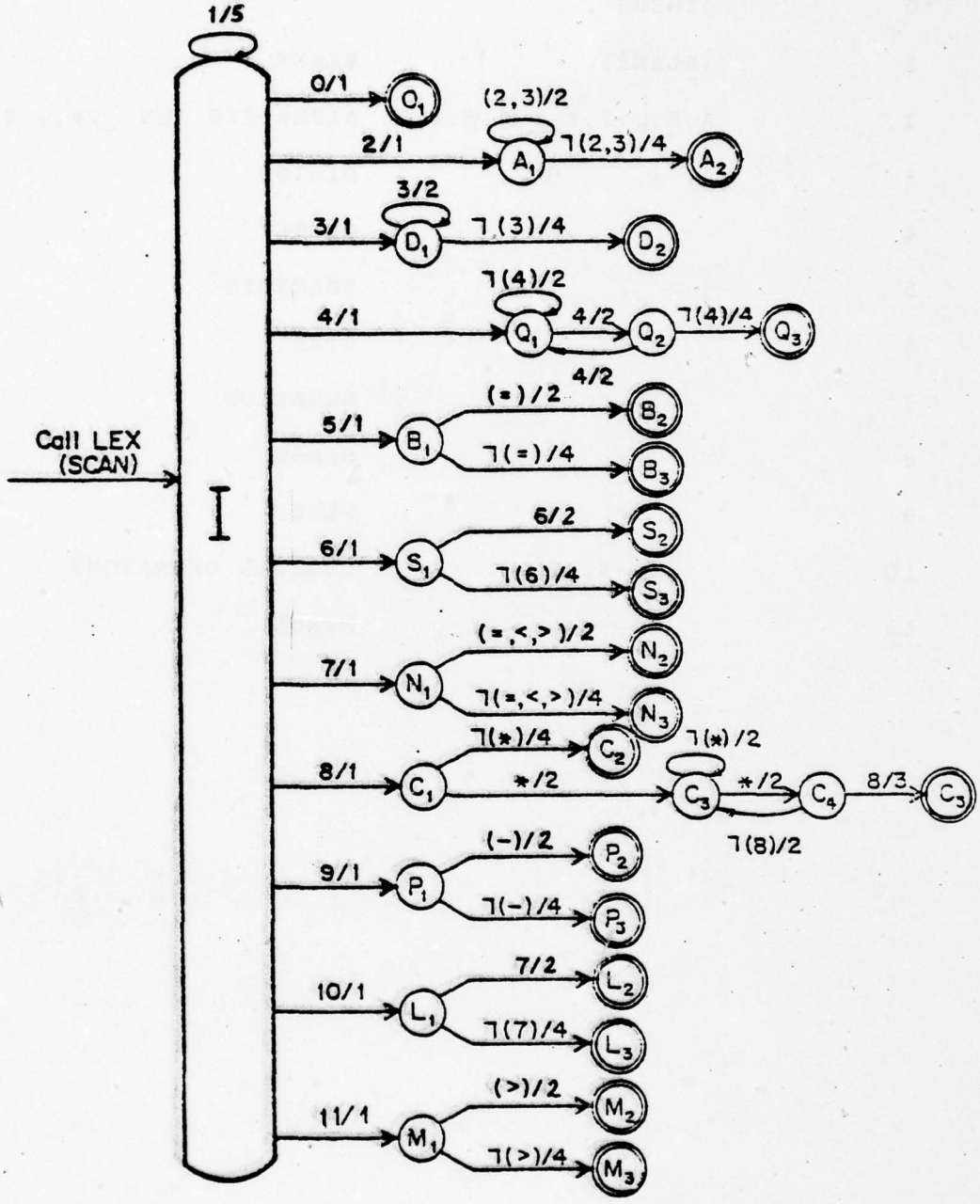


Figure 5.4 State Transition Diagram for NOPAL Lexical Analyzer

indicate the final state in which the lexical analyzer sets the type and length of the current token and then returns to the caller (namely, takes Action 3 of Table 5.3). A directed link shows the transition from a state to another after an input character is read. A notation α/β is placed alongside the link. The α before the slash indicates the character class of next input character as shown in Table 5.2. The β after the slash specifies the action to be taken before the machine changes to the next state pointed by the arrow. For example, (2,3)/2 says that if the next input character is of class 2 (an alphabet) or of class 3 (a digit), it takes action 2 (which turns out to concatenate the character to the current token) and goes to the next state. A negation sign ($\neg$) may appear before the character classes to indicate the negated input condition. For instance, $\neg$ (2,3)/4 says that if the next input character is neither an alphabet nor a digit, takes the action 4 (which turns out to decrement the input pointer) and then goes to the next state. Table 5.3 summarizes the actions taken by the lexical analyzer.

The NOPAL lexical analyzer (LEX or SCAN) is called by SAP or by a recognizer routine whenever a token is

TABLE 5.3 ACTIONS TAKEN BY LEXICAL ANALYZER OF NOPAL PROCESSOR

ACTION 1: Set next character (C) to lexical token buffer, (LEXBUFF), i.e., LEXBUF ← C;

ACTION 2: Concatenate current character to current token, i.e., LEXBUFF ← LEXBUFF || C;

ACTION 3: Set the type and length of the current token; return;

ACTION 4: Decrement current input pointer, i.e., backtrack one character;

ACTION 5: Take no action

required, using one of the two functionally equivalent calling sequences:

```
CALL LEX;                    or
IF LEXABLE THEN CALL SCAN;
```

where LEXABLE is an external 1-bit flag indicating the current enabled/disabled status of the lexical analyzer. The second calling sequence should be preferred since it actually invokes the lexical analyzer only if the analyzer is enabled, while the first sequence always invokes the analyzer. This should somewhat speed up the lexical scanning of the NOPAL because the lexical analyzer must be invoked so many times in the SAP and in the recognizer routines and because the PL/I procedure invocations need time consuming prelogues and epilogues [IBM 72]. When the lexical analyzer is invoked, it first "disables" itself by setting the flag LEXABLE to false (the returned value will always be the token until the lexical analyzer is enabled explicitly and called again). It then scans the input, extracts a token, and returns to the caller the following three items (as external variables):

(1) LEXBUFF -- the current token itself;

(2) LEXLEN -- the length (number of characters) of the token; and

(3) LEXTYP -- the type of the token. Seven types have been defined:

1) $SP: special characters, e.g. (,*,/
2) $L: logical operators, e.g. &, |, ?
3) $A: alphameric identifiers, e.g. A5_K, XYZ
4) $D: unsigned integers, e.g. 1, 345
5) $BIT: bit strings, e.g. '1001'B
6) $CHAR: character strings, e.g. 'XYZ', '123'
7) $REL: relational operators, e.g. = , < ,>

The lexical analyzer must be explicitly enabled by the calling SAP or recognizer routine, using one of the following two equivalent statements;

```
CALL LEXENAB;          or
LEXABLE = '1'B;
```

LEXENAB is an entry in the lexical analyzer which simply sets the flag LEXABLE to true when called. Again, the second assignment statement should be preferred to the first call statement for better efficiency.

In addition to LEX (or SCAN) and LEXENAB; the NOPAL lexical analyzer provides the following entry points for other services.

GETEXT: a utility to get the text of a character string.

STMT_FL: a housekeeping routine to skip input characters until the next semicolon (;) and reset the states for the SAP; called when a statement fails.

STMTEND: increments statement number counter and checks the statement and marker (;).

WARN: issues warning message during syntax analysis.

LASTREC: prints last record of source input.

POSNUMB: unsigned number recognizer routine.

In conclusion, the lexical analyzer scans the source input string and returns a token to the calling SAP or recognizer routine. As a by-product, a source specification report in which the source statements are listed with statement numbers is produced. The NOPAL lexical analyzer should be more efficient than that of the DDL or the MODEL due to the following reasons: (1) As indicated in Figure 5.4, the NOPAL lexical analyzer is implemented in a way that each class of tokens can be considered as a small finite state machine. Characters

before the current one can be "remembered", and it always returns good tokens without the need for further checking. (2) It provides ways of avoiding unwarranted invocations of PL/1 procedures. (3) In addition to the token itself, it also provides the type and the length of the token.

5.3.2 ERROR MESSAGE STACKING ROUTINES

This section describes the subroutines which place codes of error messages on a push-down stack upon recognition of incorrect syntactic units in NOPAL statements. SAP neither generates nor prints its own messages automatically. However, it expects the corresponding diagnostics to be placed on an "error stack" by the routines provided by the language-definer. During the syntax analysis, if the expected token is successfully recognized, SAP simply pops the corresponding error message code and continues. If the expected token is missing or incorrect, SAP prints the statement number and the corresponding error message which are on top of the error stack, it then pops the error message from the stack, scans for the statement end marker (;) and then continues. The error message stacking routines, the error codes, and their meanings are listed in Table 5.4.

TABLE 5.4 ERROR STACKING ROUTINES, ERROR CODES, AND MESSAGES

NAME	CODE	MESSAGE
ARGSUBS	AX-ARG	missing/invalid argument of function call or subscript of a subscripted variable
ASRT1	NOCOLN	missing colon ':'
ASRT3	AS-REL	missing relational operator in an assertion
AXER1	AX-BAD	missing/invalid arithmetic expression
	NORPAR	missing right parenthesis ')'.
BXER1	BX-REL	missing relational operator in a boolean expression
BXER2	BX-BAD	missing/invalid boolean expression
	BXLPAR	missing left parenthesis '(' after negation '¬ '
	BXRPAR	missing right parenthesis ')'.
CDER1	CDE-ID	missing/invalid connector id in conn_dim_ex
CDER2	CDENO>	missing right triangular bracket '>' in conn_dim_ex
CMNPSFF	DGCMPF	missing/invalid affected component
CMPSLØP	DG-LOP	'&' and '|' operators mixed
	DGCMPF	missing/invalid affected component
COLON	NOCOLN	missing colon ':'.
CONJ5	CJ-REL	missing relational operator in a

TABLE 5.4 (continued)

NAME	CODE	MESSAGE
		conjunction
	NORPAR	missing right parenthesis
DCLER1	DCL-ID	missing variable ID in declaration
DIAGER2	NO-MSG	missing 'MESSAGE' after 'OPERATOR' in keyword diagnosis definition
	NOCOLN	missing colon ':' after 'OPERATOR MESSAGE' in keyword diagnosis definition
DIAG1	DG-LBL	missing diagnosis label
FDER1	FDEBAD	missing/invalid function dimension expression
	FDRPAR	missing right parenthesis in func-dim-ex
FNER1	FNTYPE	missing/invalid function type
GETASRT	AS-BAD	missing/invalid assertion
IFCOND	IFTHEN	missing THEN in an IF-clause
KEYEQ	KEY-EQ	missing equal sign '=' after a keyword
LOGIC2	LG-OPR	missing/invalid logical operator
	DG-LBL	missing diagnosis label after logical operator

TABLE 5.4 (continued)

NAME	CODE	MESSAGE
MSGER1	MSGLBL	missing message label
MSGER2	KEY-EQ	missing equal sign '=' after 'ALIAS'
	MSNAME	missing message synonym after '='
	MSCOMA	missing comma ',' after message synonym
OPMSG3	DGMSGA	missing/invalid message argument in other parameters of a diagnosis
OPRPS3	DG-VAR	missing/invalid operator response variable ID
RPAR	NORPAR	missing right parenthesis ')'
SAME1	BR-LBL	missing/invalid back-reference stim/meas lable
SPECERR	SP-BAD	invalid specification statement
TBLER1	KEYBAD	missing/invalid keyword
	KEY-EQ	missing equal sign '=' after a keyword
	KEYTXT	missing/invalid text after '='
TBLER2	NO-ID	missing identifier
TSMER1	TSM-ID	missing parent label after '('
	NORPAR	missing right parenthesis ')'
TXTBAD	MSGTXT	missing/invalid message text

5.3.2.1 WHERE THE ERROR MESSAGE STACKING ROUTINES ARE USED

Preceding every required terminal symbol or non-terminal associated with a recognizer routine in the EBNF/WSC production rules of NOPAL, a routine name must be inserted to stack an error message code in case the token is missing or incorrect. A non-terminal associated with a recognizer routine is a non-terminal which is eventually defined to reference a recognizer routine (recognizer routines and non-terminals associated with them are further discussed in Section 5.3.3). To illustrate, consider the EBNF statement 102 of Figure 3.1:

```
< MESSAGE_DEFINITION >::=  MESSAGE  < MESSAGE_LABEL >[:]
        [ALIAS = < SYNONYM >,] [TEXT = ] < MESSAGE_TEXT >:
```

The corresponding production rules in the EBNF/WSC of NOPAL (lines 146-151, 3, and 59 of Figure 5.2) become:

```
< MESSAGE_DEFINITION > ::= < MESSAGE >/MSGER1/
                           < MESSAGE_LABEL >
    /SVLBL/  [:]  [ALIAS/MSGER2/ = < SYNONYM >/SVSYN/,]
    [TEXT/KEYEQ/ = ] < MESSAGE_TEXT >/STMSG/ /STMTEND/
< MESSAGE >::= /MESSAGE/
< SYNONYM >::= < IDENTIFIER >
< LABEL > ::= /LABEL/
< IDENTIFIER > ::=  /NAMEREC/
<MESSAGE_LABEL > ::= < LABEL >
```

In the above example, < MESSAGE >, < MESSAGE_LABEL > , and < SYNONYM > are non-terminals associated with the recognizer routines MESSAGE, LABEL, and NAMEREC respectively. MSGER1, MSGER2, and KEYEQ are error message stacking routines. The two non-terminals < MESSAGE_LABEL > and < SYNONYM >, and the three terminals "=", ",", and another "=" are mandatory, and hence the corresponding five error message codes are required in this production rule. The routine MSGER1 has been inserted before < MESSAGE_LABEL > to stack the missing message label error; the routine MSGER2 has been inserted after the first element ALIAS of an optionality group to stack three error messages for the three corresponding non-optional symbols "=", < SYNONYM > and ",". The third error stacking routine KEYEQ stacks an error message for the mandatory equal sign ("=") after the keyword TEXT in another optionality group. Note that no error messages have been stacked for non-terminals such as < MESSAGE_TEXT > in the above production. An appropriate error message for each non-optional terminal symbol or non-terminal associated with a recognizer routine is expected to be stacked in the lower-level grammar tree such as the production rule for < MESSAGE_TEXT >. Note also that there is no error

message stacked for the very first element of an optionality group. Similarly of a production, unless the production is mandatory.

5.3.2.2 HOW TO WRITE ERROR MESSAGE STACKING ROUTINES

Error message stacking routines have the following general form in the current NOPAL implementation:

```
< ROUTINE_NAME >:  PROCEDURE;
    DCL ERRORS(n) CHARACTER(6) STATIC INITIAL
       ('c1', 'c2', ... 'cn');
    CALL $PUSH(ERRORS);
    RETURN;
  END < ROUTINE_NAME >;
```

Where < ROUTINE_NAME > is the name of an error message stacking routine, n(> 1) is the total number of error codes to be stacked in this routine, and c_1, c_2,..., c_n are the actual error codes of six characters each. Note that the codes will be pushed down the error stack in the reverse order that they appear. That is, c_n will be at the bottom, while c_1 at the top. For the two error stacking routines MSGER1 and MSGER2 in the above example, the corresponding PL/I codes become:

```
MSGER1:  PROCEDURE;
    DCL MSGLBL(1) CHAR(6) STATIC INIT('MSGLBL');
```

```
      CALL $PUSH(MSGLBL);
      RETURN;
   END MSGER1;                          and
   MSGER2:  PROCEDURE
      DCL MSGERR(3) CHAR(6) STATIC INIT
         ('KEY-EQ', 'MSNAME', 'MSCOMA');
      CALL $PUSH(MSGERR);
      RETURN;
   END MSGER2;
```

5.3.3 RECOGNIZER ROUTINES

A recognizer routine is a PL/I procedure which recognizes a certain class of tokens from the input string. It is a function which returns a 1-bit value of 1 or 0, representing true or false respectively, depending upon the success or the failure of recognition. Recognizer routines are primarily used to speed up the recognition process of SAP; they are often employed when it would be clumsy and time-consuming to have SAPG generate necessary code to do the required analysis. For example, to recognize a character string as an identifier (i.e., a name) the following two EBNF productions (statements 18 and 19 of Figure 3.1) could be directly used in the EBNF/WSC:

< IDENTIFIER > ::= < LETTER >[< TAIL >]*

< TAIL > ::= < LETTER > | < DIGIT >

It would require as many iterations as the number of characters in the name. Instead, a recognizer NAMEREC has been used (in the third line < IDENTIFIER > ::= /NAMEREC/ of the EBNF/WSC) to analyze it in a single pass and even to check the length of the name. NOPAL keywords consisting of more than five characters are also implemented as recognizers, each of which recognizes as a good keyword a set of all names having the same prefix of four characters. All recognizer routines in the NOPAL are enumerated in Table 5.5.

5.3.3.1 HOW TO DENOTE AND REFERENCE RECOGNIZER ROUTINES IN EBNF/WSC

Given a production rule "L ::= R", the non-terminal L is called the left-part of the production; the R, consisting of one or more terminals or non-terminals, is called the right-part of the production. If R is a non-terminal alone, the production is a singular production.

A recognizer routine is represented by an EBNF/WSC production which involves a stand-alone subroutine call as the right-part, i.e., a production called recognizer production, of the following general form;

< L > ::= /RECOGNZ/

where < L >is a non-terminal and RECOGNZ is the name of

TABLE 5.5 RECOGNIZER ROUTINES WITH EXAMPLES

NAME	WHAT IT RECOGNIZES	EXAMPLES
ANDOROP	AND (&) and OR (|) logic operators	&, |
ARROW	Arrow (→)	→
CHARSTR	Character strings	'XYZ', '101'
DIMREC	Dimensions (units)	VOLT, OHM
EXPONET	Exponentiation operator (**)	**
INTEGER	Integers (signed or unsigned)	12, +34, -56
LABEL	Labels (identifiers or unsigned integers)	A1_J, 345
LOGICOP	NOPAL logical operators	&, |,*,?,|
NAMEREC	Identifiers	@X2_B, XYZ
NUMBER	Numbers (signed or unsigned)	3.5, -1.0E+70
OR_OP	OR (|) logic operator	|
PLUSMIN	Range operator (+-)	+-
POSTING	Unsigned integers	1234
POSNUMB	Unsigned numbers	12,3.5,1.0E-70
RELREC	Relational operators	<, =, >, =, <=
STRREC	String constants (character or bit)	'XYZ', '101'B
TIMEEDM	Time-dimensions	MIN, SEC, MSEC

TABLE 5.5 (continued)

The following are NOPAL keyword-stem recognizers. If a keyword is more than five characters long, only the first four characters will suffice. "..." denotes zero or more alphameric characters.

NAME	WHAT IT RECOGNIZES	EXAMPLES
ASSERT	ASSE..., ASRT...	ASSERTION,ASRT,ASSE
ATE_PNT	ATE_...	ATE_POINT,ATE_PT,ATE_P
COMMENT	COMM...	COMMENTS, COMM
COMPFL	COMP...	COMP_FAIL,COMP,COMPONENT
CONJUNC	CONJ...	CONJUNCTION, CONJ
CONNECT	CONN...	CONNECTION, CONN, CONNECTORS
COOPERA	COOP...	COOPERATION, COOP
DIAGNOS	DIAG...	DIAGNOSIS, DIAG
EXCEPT	EXCE...	EXCEPT, EXCE
FAILURE	FAIL...	FAILURE, FAIL
FUNCTION	FUNC...	FUNCTION, FUNC
MEASURE	MEAS...	MEASUREMENT, MEAS
MESSAGE	MESS...	MESSAGE, MESS
PARAMET	PARA..., PARM...	PARAMETER, PARM, PARA
PROTECT	PROT...	PROTECTION, PROT
RESPONS	RESP...	RESPONSE, RESP
SPECIF	SPEC...	SPECIFICATION, SPEC

TABLE 5.5 (continued)

NAME	WHAT IT RECOGNIZES	EXAMPLES
SRC_TGT	SOUR...;TARG...	SOURCE, SOUR:TARGET, TARG
STIMULI	STIM...	STIMULI, STIM, STIMULUS
UUT_PNT	UUT_...	UUT_POINT,UUT_PT,UUT_P

the recognizer routine to be called. In other words, if a subroutine call appears in a production as the right-part, the subroutine is a recognizer routine. For example, the NAMEREC is a recognizer routine in the following production:

< IDENTIFIER >::= /NAMEREC/

There is a class of non-terminals which are associated with a given recognizer routine. A non-terminal < N >is said to be in this class if < N >is the left-part of the recognizer production, or recursively < N >is the left-part of a singular production whose right-part is a non-terminal in this class. In other words, a non-terminal is said to be associated with a recognizer routine if it is eventually defined to reference the recognizer routine. For example, < SYNONYM > and < IDENTIFIER > in the following two productions are both associated with the recognizer routine NAMEREC.

< SYNONYM > ::= < IDENTIFIER >

< IDENTIFIER > /NAMEREC/

As far as error message stacking is concerned, non-terminals associated with recognizer routines play the same role as terminal symbols do, hence they are treated exactly in the same way in preparing EBNF/WSC.

5.3.3.2 HOW TO WRITE RECOGNIZER ROUTINES

Recognizer routines are PL/I procedures which return a bit string of length 1. They must perform the necessary lexical functions that SAP does. Normally, upon entry to such a routine, the lexical analyzer (LEX or SCAN) is called to get a lexical unit (token) to be analyzed. More than one token may need to be obtained to complete analysis. After the necessary analysis, if recognition is successful, the lexical analyzer must be "enabled" by setting LEXABLE to '1'B (or calling LEXENAB, see Section 5.3.1) and '1'B (true) must be returned. Otherwise, '0'B (false) must be returned.

As indicated in Section 5.3.1, in addition to the token available in LEXBUFF, the type and the length of the token are also available in LEXTYPE and LEXLEN respectively. Seven token types are defined in NOPAL. Three entry points LEX, SCAN and LEXENAB, and a lock switch LEXABLE are defined in the lexical analyzer (see Section 5.3.1). All of these are PL/I external variables, and hence can be accessed for analysis in preparing recognizer routines. To illustrate these features, the recognizer routine NAMEREC appears as follows:

```
NAMEREC:  PROCEDURE RETURNS (BIT(1));
   DCL LEXBUFF CHAR(31) VAR EXT, /* TOKEN BUFFER */
       (LEXTYP, LEXLEN) FIXED BIN EXT,
            /* TOKEN TYPE & LENGTH */
       LEXABLE BIT(1) EXT, /* LEX lock switch */
       $A FIXED BIN EXT; /* TOKEN TYPE: ALPHAMERIC */
   DCL (T INIT('1'B), F INIT('0'B)) BIT(1) STATIC;
   IF LEXABLE THEN CALL SCAN; /* OR CALL LEX: */
   IF LEXTYP  = $A THEN RETURN(F);
   IF LEXLEN > 12 THEN
      DO; LEXLEN = 12;
          LEXBUFF = SUBSTR(LEXBUFF, 1, LEXLEN);
          CALL WARN(', NAME/INTEGER TOO LONG.
                                TRUNCATED.');
          END;
      LEXABLE = T; /* ENABLE LEX; OR CALL
                                    LEXENAB;  */
      RETURN(T);
END NAMEREC;
```

The first two statements are variable declarations. Then the routine calls the lexical analyzer, via entry SCAN, to get a token. If the token is not alphameric then F(false) is returned. Otherwise, it continues to check the length of the token to be no greater than 12.

Finally, the lexical analyzer is enabled and T(true) is returned.

5.3.4 ENCODING/SAVING/STORING ROUTINES

Encoding/saving routines are used to encode some of the NOPAL syntactic units and save them into an internal representation. Although some entities such as names provided by a user are kept intact in internal form, many of the descriptions and attributes are encoded, compacted and saved for more efficient processing later. For example, there are fourteen types of NOPAL statements and they are encoded internally as fourteen integers. Storing routines are specified at the end of source statements. They gather information in the statements, and in turn call the STORE subsystem (to be discussed in Section 5.4) to store the statements for later processing. Each NOPAL statement corresponds a tree-like data structure (implemented as a PL/1 based structure), which is used to store all of the information about the statement. The encoding, saving, and storing routines all together are responsible for creating such statement data structures, collecting and filling in all the information, and passing them to the STORE subsystem to update the directory and properly link the storage keys for accessing the statements. The data structure of all NOPAL statements in conjunction with the corresponding PL/1 based

structure declarations are presented in Appendix A. The STORE/RETRIEVE subsystem is discussed in Section 5.4. The encoding, saving and storing routines are summarized in Table 5.6 in the groups of statements.

5.3.4.1 WHERE THE ENCODING/SAVING/STORING ROUTINES ARE USED IN EBNF/WSC

The encoding and saving routine names are inserted in the EBNF/WSC statements whenever there is a need to encode and save a syntactic unit for later storing. Each routine is automatically invoked by SAP after the successful recognition of the syntactical unit immediately preceding the routine call. A storing routine name is inserted at the end of each NOPAL statement to collect the information about the statement and to invoke the STORE routine of the STORE/RETRIEVE package for storing the statement. To illustrate, take the following EBNF/WSC statement (line 51 of EBNF/WSC for NOPAL):

```
[NOPAL] < SPECIFICATION >[ < SPEC_NAME >/SVLBL/]/
                                              /STSPEC/
```

where < SPECIFICATION > is associated with a recognizer for keyword SPECIFICATION; < SPEC_NAME > is associated with a recognizer for a label (in this case, a name for NOPAL specification) (see lines 57 and 58 of the EBNF/WSC). The SVLBL subroutine call is inserted immediately after < SPEC_NAME > in order to save the label.

TABLE 5.6 ENCODING, SAVING, AND STORING ROUTINES

1.	STATEPT	---	Stores ATE_CONNECTION_POINT statement
	SCOMPT	---	Saves comment
	SVPTID	---	Saves matching UUP_POINT ID's
	SVSEQ#	---	Saves internal table entry sequence number
	SVSYN	---	SAVES synonym of the ATE connecting point ID
	SVLBL	---	Saves ATE connecting point ID
2.	STCOMP	---	Stores component/failure statement
	SVPARML	---	Saves parameter list of failure function
	SCOMT	---	Saves comments
	FLSFF	---	Saves failure function
	FLSIDX	---	Saves failure index
	SVCMPFL	---	Saves component failure seq# for component protection
	SVLBL	---	Saves the sequence number for this component failure
	UUTCMPF	---	Allocates component-fail entry and saves component ID
	SVSYN	---	Saves synonym of the component ID
	PMSID	---	Saves parameter ID
3.	STDIAG	---	Stores diagnosis statement
	DIAG1	---	Allocates and initializes the diagnosis entry

TABLE 5.6 (continued)

	SVLBL	---	Saves diagnosis label
	SPARM	---	Saves other parameters
	CMPRLOP	---	Allocates entry for affected components
	CMPSID	---	Initiates and saves components
	SVCMPFL	---	Saves component-failure sequence number
	MASTR	---	Saves message parameter: string constant
	CMPSLOP	---	Saves Logical operator (& or |)
	MANUM	---	Saves message parameter: number
	MAVAR	---	Saves message parameter: variable
	OPRPS1	---	Saves operator response Y/N (i.e., ?)
	OPRPS2	---	Saves variables of operator response
	TIME1	---	Saves value of timing
	TIME2	---	Saves dimension of timing
	STYP	---	Saves message type
4.	STEND	---	Store END statement
	SVLBL	---	Saves label
5.	STFUNC	---	Stores ATE_function statement
	SVSEQ#	---	Saves internal table entry sequence number
	SVSYN	---	Saves synonym of ATE function ID
	FNSTYP	---	Saves function type
	ATEFUNC	---	Allocates function entry and saves function ID

TABLE 5.6 (continued)

	FN#PINS	---	Saves number of connecting pins of S/M functions
	FNSVAL	---	Saves values returned
	PMSTYP	---	Saves parameter type
	FNSCF	---	Saves cooperation functions
	SCOMPT	---	Saves comments
6.	STLOG	---	Stores logic statement
	SVLBL	---	Saves label
	SVLBL2	---	Saves label (secondary) of the parent test module
	LOGID	---	Saves logic entry ID
	SLOPLBL	---	Saves logical operator and diagnosis label
7.	STMEAS	---	Stores measurement statement
	SVLBL	---	Saves label (primary) of the measurement
	SVLBL2	---	Saves label (secondary) of the parent test module
8.	STMSG	---	Stores message statement
	SVLBL	---	Saves label
	SVSYN	---	Saves Synonym of the message label
	TXTCH	---	Saves character-string message text
9.	STSPEC	---	Stores specification statement
	SVLBL	---	Saves label

TABLE 5.6 (continued)

10.	STSTIM	---	Stores stimuli statement
	SVLBL	---	Saves label (primary) of the stimuli
	SVLBL2	---	Saves label (secondary) of the parent test module
11.	STTEST	---	Stores test statement
	SVLBL	---	Saves label
12	STUUTPT	---	Stores UUT-connection-point statement
	SVSEQ#	---	Saves internal table entry sequence number
	SYSYN	---	Saves synonym of the UUT connecting point ID
	SVCNTYP	---	Saves connector type
	SVCNPT	---	Saves connector point
	GETLMT	---	Gets protective limit entry
	PMSDM	---	Saves parameter dimension
	PMSHL	---	Saves upper limit of parameter
	PMSLL	---	Saves lower limit of parameter
	PMSRPT	---	Saves parameter reference point
	SCOMT	---	Saves comments
	PMSID	---	Saves parameter ID
	PMSTY	---	Saves parameter type
	UUTPNT	---	Allocates UUT point entry and saves UUT point ID

TABLE 5.6 (continued)

13.	STWAVFM	---	Stores conjunction waveform
	CONJ1	---	Allocates conjunction entry and stores conjunction label
	SVLBL	---	Saves label (primary) of the conjunction
	SVLBL2	---	Saves label (secondary) of the parent stimuli or measurement
	STRIPT	---	Allocates and saves simple conjunction
	CONJ5	---	Allocates connector entry
	CJSFDE	---	Saves function-dimension-expression
	BRSLBL	---	Saves back reference label for conjunction
	DCLSVAR	---	Saves a variable in variable-declaration
	DCLSUBS	---	Saves subscripts of variable-declaration
	DCLSID	---	Saves and verifies the IDs of variable-declaration
	CDESID	---	Saves connector ID
	CDESDM	---	Saves connector dimension
	FDESDM	---	Saves dimension in function-dimension-expr
	FDES1	---	Saves a token in func-dim-ex
	SETFDE	---	Gets a stack for new func-dim-ex
	FDESFDE	---	Saves a nesting func-dim-ex
	FDESMOD	---	Save modifier for func-dim-ex

TABLE 5.6 (continued)

14. STWAVFM --- Stores assertion waveform

ASRT1 --- Allocates assertion entry and stores assertion label

SVLBL --- Saves label (primary) of the assertion

SVLBL2 --- Saves (secondary) of the parent stimuli or measurement

ASEXPS --- Saves expressions before and after the relational operator of an assertion

ASREL --- Saves relational operator of an assertion

ASRANGE --- Saves range of an assertion

ASPC --- Sets the percentage of the range

DCLSUBS --- Saves subscripts of variable-declaration

DCLSID --- Saves and verifies IDs of variable-declaration

GETASRT --- Allocates and saves simple assertion

15. The following are the encoding and saving routines which are used in arithmetic and Boolean expressions.

AREXSAV --- Concatenates an arithmetic sub-expression

AREXS1 --- Saves a token for arithmetic expression

AREXSVF --- Saves a variable or function call

SETVF --- Allocates a stack for a variable or function call

TABLE 5.6 (continued)

VFSO	---	Sets subscript switch and saves a token for variable or function call
VFS1	---	Saves a token for variable or function call
VFS2	---	Gets and saves string constant for a function call
VFSAX	---	Saves arithmetic expression in a variable subscript or a function argument
IFCOND	---	Saves the condition part of an IF clause
SETBEXP	---	Allocates a stack for Boolean expression
BEXPS1	---	Saves a token for Boolean expression
BEXPSAX	---	Saves an arithmetic expression in Boolean expression
BEXPSBX	---	Concatenates a Boolean sub-expression
SETAREX	---	Allocates a stack for arithmetic expression

At the end of the statement, STSPEC subroutine call is included to store the statement.

5.3.4.2 HOW TO WRITE ENCODING/SAVING/STORING ROUTINES

These are subroutine-type PL/1 procedures and ought to be so prepared. For each source statement, a storing routine and a set of encoding/saving routines are required to allocate data structure, encode and/or save necessary information, accumulate names (also types) of the storage keys, and finally pass these to the STORE subsystem for updating the directory and storage entries in the simulated associative memory. The STORE and the associative memory are discussed in Section 5.4. For instance, the two routines SVLBL and STSPEC in the above mentioned statement are as follows:

```
STSPEC:  PROCEDURE;
    DCL SPEC# FIXED BIN EXT; /* KEY & STMT TYPE:  SPEC */
    DCL STMT_NO FIXED BIN EXT, /* STMT NO. */
        LEXBUFF CHAR(31) VAR EXT; /* TOKEN BUFFER */
    DCL 1 STORAGE (*) EXT CTL,
        2 NAMES CHAR(12) VAR,  /* KEY NAMES */
        2 TYPES FIXED BIN;  /* KEY TYPES */
    DCL N# FIXED BIN EXT;   /* NO. OF KEYS */
    DCL SPECHAR CHAR(12) VAR INIT ('SEPECHARS');
```

```
    DEC 1 SPEC BASED (DP), /*SPEC DATA STRUCTURE */
          2 TYPE FIXED BIN, /* STMT TYPE */
          2 STMT# FIXED BIN; /*STMT NO. */
    ALLOCATE SPEC;
    SPEC.TYPE=SPEC#;  SPEC.STMT# = STMT_NO;
    IF N# = 1 THEN
       DO; TYPES(1) = SPEC#; SPECNAM = NAMES(1); END;
    ELSE DØ; N# = 1;
            TYPES(1) = -SPEC#; NAMES(1) = SPECNAM;
         END;
    CALL STORE (DP);
    RETURN;
SVLBL:  ENTRY;   /* SAVE LABEL */
    N# = 1;   NAMES(1) LEXBUFF;
    RETURN;
END STSPEC;
```

The SVLBL routine simply saves the label and sets the counter (N#) of storage keys to 1. The STSPEC routine first allocates a data structure, encodes the statement type as SPEC# (it turns out to be integer 1, representing "specification") and saves the statement number. Then, it checks if the label has been provided by the user. If the label has not been specified, the default label $SPEC00000 will be assigned and a flag

set. The key type is set to SPEC# either way. Finally, the STORE routine from the STORE/RETRIEVE package is invoked to put the statement in the simulated associative memory.

5.3.5 LOCAL SEMANTICS CHECKING ROUTINES

Some of the local semantics of the NOPAL statements can be checked during the syntax analysis phase. Such semantics checking routines can check that a condition, such as a range, on a syntactic unit is locally correct, something cannot be done through syntax specification only. These routines do not and cannot check the overall correctness, consistency or completeness of the whole NOPAL specification, a task which will be performed by the later specification analysis phase of the NOPAL processor. To illustrate, part of a EBNF/WSC production (lines 194-197), corresponding to the EBNF statement 125 of Figure 3.2 is shown as follows:

```
< PROTECTIVE_LIMITS > ::= ... [ < MAX_LIMIT >/PMSHL/]
                                                [,
                                        [ < MIN_LIMIT >/PMSLL/]
                                                ...]
```

It specifies the upper and lower protective limits for a UUT connecting point. After the upper limit < MAX_LIMIT >

the routine PMSHL is invoked to save it. Similarly, after the lower limit < MIN_LIMIT > the routine PMSLL is called to save it and at the same time to check it to be no greater than the upper limit. The statement semantics checking routines are listed in Table 5.7.

5.3.5.1 WHERE THE LOCAL SEMANTICS CHECKING ROUTINES ARE USED

These routines for checking local statement semantics are optional and hence at the discretion of the language definer. They are inserted in the EBNF/WSC productions after the appropriate syntactic units so that upon the successful recognition of these units by SAP, they are invoked to check locally that the statement semantics is correct. Normally, these routines are coordinated or even combined with some encoding, saving, or storing routines. For instance, in the above example, the routine PMSLL is inserted immediately after the specification of lower protective limit < MIN_LIMIT > , first to save it and then to make sure the value is smaller than the upper limit which was saved by the routine PMSHL.

5.3.5.2 HOW TO WRITE LOCAL SEMANTICS CHECKING ROUTINES

The routines are subroutine-type PL/1 procedures. Inside such a routine, the lexical token and all the

TABLE 5.7 LOCAL SEMANTICS CHECKING ROUTINES

NAME	WHAT IT CHECKS
CJREL	checks relational operator is '=' in connector-dimension-expression.
CKSTR	checks a string constant never appears in a subscripted variable.
CMPSLOP	checks all logical operators in the affected components of a diagnosis are either OR (|) or AND (&) but not mixed.
DCLSID	checks each variable id in a SOURCE or TARGET declaration has appeared in the same conjunction or assertion statement.
INTEGER	recognizes integers and checks the number of digits in an integer.
LABEL	recognizes labels (i.e., identifiers or integers) and checks their length (number of characters)
NAMEREC	recognizes identifiers (i.e., names) and checks their length.
NUMBER	recognizes any numbers and checks their length.
PMSLL	saves lower protective limit for a UUT connection point and checks the lower limit is not greater than the upper limit.
POSINTG	recognizes unsigned integers and checks their length
POSNUMB	recognizes unsigned numbers and checks their length

information encoded and/or saved previously by other routines may be accessed for the purpose of checking local semantics. To illustrate, again consider the above-mentioned < PROTECTIVE_LIMITS > example. The semantics checking (and also saving) routine PMSLL appears as follows:

```
PMSLL:   PROCEDURE;
   DCL   LEXBUFF CHAR(31) VAR EXT; /*TOKEN BUFFER*/
   DCL   1 LIMIT BASED(TP),  /* DATA STRUCTURE */
           2 DIM CHAR(12),   /* FOR LIMITS */
           2 MAX DEC FLOAT,
           2 MIN DEC FLOAT,
           2 REF_PT FIXED BIN;
   LIMIT.MIN = LEXBUFF;
   IF LIMIT.MIN > LIMIT.MAX THEN
     DO; LIMIT.MIN = - 1E+75;
         CALL WARN(', MIN.LIMIT > MAX.LIMIT ... ');
     END;
     RETURN;
END PMSLL;
```

The routine first saves the lower protective limit in a data structure. Then it compares the lower limit with the upper limit, which was saved by another routine

PMSHL. If the lower limit were found greater than the upper one, the lower limit would be set to a negative number (-10**75) and a warning message would be sent to the user.

5.3.6 HOUSEKEEPING ROUTINES

There are a few "housekeeping" type subroutines, most of which need not be written by the language definer because they are provided by the SAPG system. However, their names need to be properly included in the EBNF/WSC. Two of these subroutines appear in the very first EBNF/WSC production of NOPAL (in fact, any language using the SAPG must be written in a similar way), which are reproduced as follows;

```
< NOPAL_SPECIFICATION >::=  [ < NOPAL_STMTS >/CLRERRF/]*
                                    /STMT_FL/
                                      < NOPAL_SPECIFICATION >
```

< NOPAL_SPECIFICATION > is the goal symbol of the NOPAL language. It is defined as zero or more < NOPAL_STMTS >, each of which turns out to be further defined as one of the NOPAL statements. After one statement is recognized, the next statement is attempted; this process repeats until end of the input text. SAPG requires that the subroutine CLRERRF ("Clear Error Flag"),

which resets the error flag for the generated SAP, be called at the end of each statement [FRE 72]. During the recognition process, if none of < NOPAL-STMTS > statement types is found to match, the above production indicates to branch to the other subroutine STMT_FL ("Statement Fail"). This routine scans the input text for the statement end marker (";") and resets the conditions for SAP to begin processing the next statement. Finally, the < NOPAL_SPECIFICATION > causes SAP to attempt recognizing the next statement recursively. The CLRERRF has been built in the FAILMAN package of the SAPG system and hence is always applicable to all languages; the STMT_FL has been written as an entry of the lexical analyzer and usually need not be rewritten. These two routines appear in the first production only, as explained.

The next two housekeeping routines appear elsewhere. They have been implemented as two entry points of the lexical analyzer, but they are applicable to other languages. STEND (on line 56 of EBNF/WSC) is called at the end of the source input (i.e., after "END" is read) to print the last record of input. STMTEND is invoked after each NOPAL statement in order to increment the statement number for the lexical analyzer.

The above mentioned four subroutines (CLRERRF, STMT_FL, STEND, and STMTEND) are explicitly inserted in the EBNF/WSC properly. To be precise, the SAPG-generated SAP also uses other five housekeeping routines (each one is prefixed with "$"): $MARK, $POPF, "SUCCESS, $FAIL, and $PUSH [RAM 73]. Except the $PUSH, which is also used to push error codes in error-stacking routines (as indicated in Section 5.3.2), they are not useful to, and should not concern, the language definer. All housekeeping routines with their functions are summarized in Table 5.8.

5.4 THE STORE/RETRIEVE SUB-SYSTEM AND ASSOCIATIVE MEMORY

5.4.1 Introduction

In order to facilitate the SAPG system with a general-purpose mechanism for storing source statements and for later retrieval, the following subsystem has been implemented and included in the NOPAL Processor. This is modified from the MODEL's string storage and retrieval subsystem [RIN 76] and the routines are completely rewritten to meet NOPAL's special requirements and to increase processing efficiency. The subsystem consists of two types of routines.

TABLE 5.8 HOUSEKEEPING ROUTINES

NAME	WHAT IT DOES
CLRERRF	clear "error"flag for the SAP after each statement to continue processing next statement
STEND	prints last record by calling LASTREC (in lexical analyzer); is invoked upon end of source input
STMTEND	checks statement end marker (;) and increments the statement number; called at end of each statement
STMT_FL	scans for next statement end marker (;) and resets for the SAP; called when recognition of a statement fails
$FAIL	prints error message on top of the error stack when a local error occurs
$MARK	marks the beginning of errors for a new production; called upon entry to each production and is done by pushing blank error code onto the error stack
$POPF	pops the top entry of the error stack; called after successful recognition routine reference
$PUSH	pushes one or more error codes onto the error stack; called explicitly from $MARK or error-

TABLE 5.8 (continued)

NAME	WHAT IT DOES
	stacking routines
$SUCCESS	restores the error stack to the way it was before a production was invoked; called upon termination of a production, successfully or not.

(1) STORE for storing source language strings gathered during the phase of syntax analysis; and

(2) RETRIEVE for retrieving stored source language statements, and for accessing "directory entries"; the former is through an entry RETREVS and the latter another entry RETREVD.

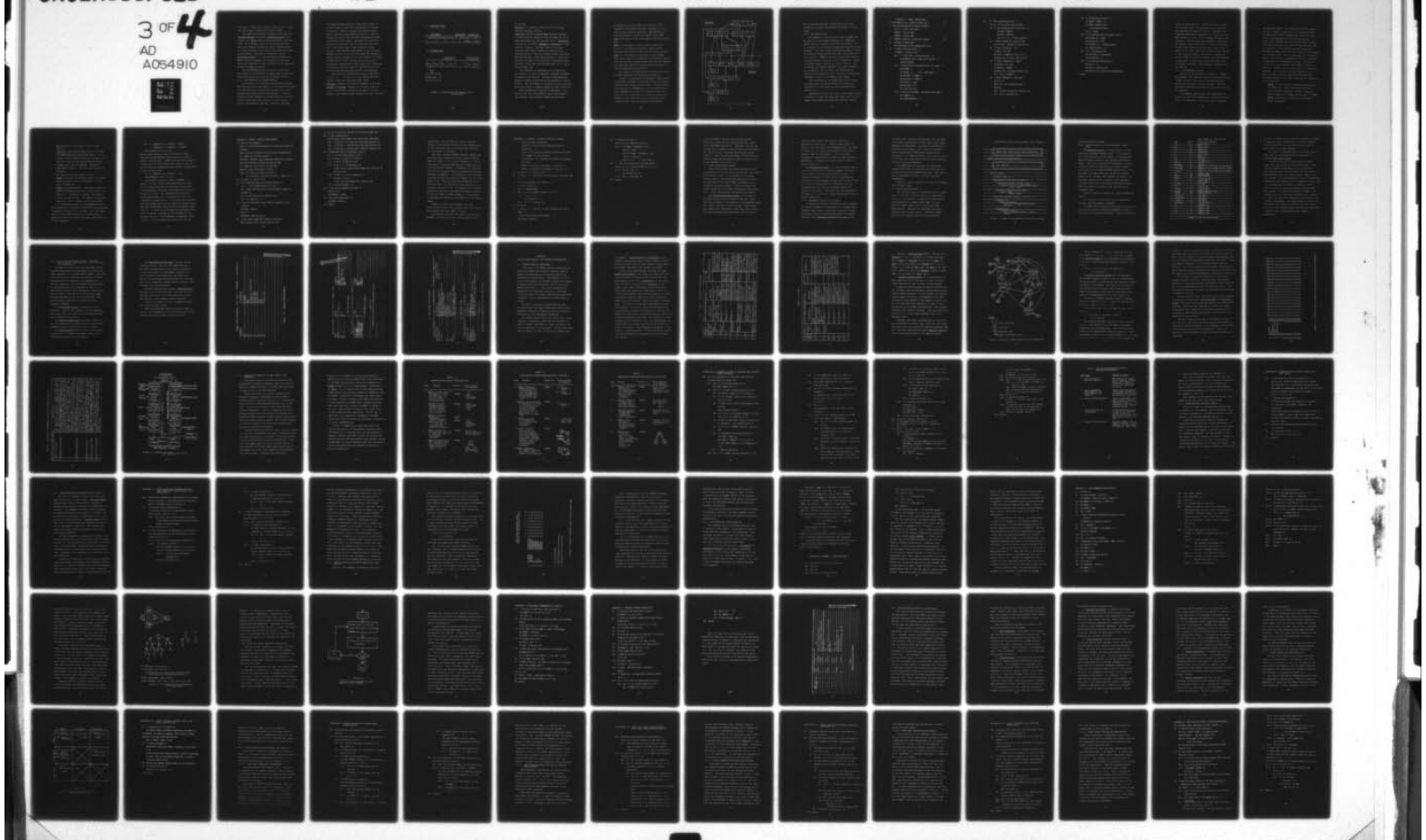
In the NOPAL system, a "name" (i.e., a string of characters to identify some entity) may be associated with more than one "type". Basically a type may be interpreted to designate a certain class of entities. Thus a name and a type together uniquely identify an entity. For example, a name X of type 2 (representing test modules) identifies a test module, while the same name X of type 5 (representing diagnoses) may also be used to denote a diagnosis. There are sixteen types of names and/or statements defined in NOPAL system, as enumerated in Table 5.9. One of the advantages of allowing a name to be able to represent multiple types of entities is that the user has more freedom in giving names as long as each name is unique in a given class. For instance, unsigned integers can be used as names in most cases, hence the user may choose to use such sequence numbers to identify most entities without inventing

TABLE 5.9 TYPES OF NAMES AND STATEMENTS IN NOPAL

TYPE #	MNEMONIC	CLASS OF ENTITIES REPRESENTED
1	SPEC#	NOPAL specification label/statement
2	TEST#	Test module label/statement
3	STIM#	Stimulus label/statement
4	MEAS#	Measurement label/statement
5	DIAG#	Diagnosis label/statement
6	MSG#	Message label/statement
7	LOGIC#	Logic label/statement
8	CONJ#	Conjunction label/statement
9	ASRT#	Assertion label/statement
10	COMP#	UUT component identifier (id)
11	CMPFL#	Component-failure (i.e. affected component) id/statement
12	UUTPT#	UUT connection point id/statement
13	ATEPT#	ATE inter-connection point/id statement
14	FUNC#	Function id
15	VAR#	Variable id
16	END#	End statement

AD-A054 910 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 14/2
AUTOMATIC TEST PROGRAM GENERATION.(U)
MAR 78 Y K CHANG DAAA25-75-C-0650
UNCLASSIFIED 77-02 ECOM-75-0650-F-1 NL

3 OF 4
AD
A054910



many names. Consequently from the user's point of view the total number of names can be much reduced.

The STORE routine stores strings in main memory as "storage entries" and builds "directory entries" in a directory of "keys" (each is associated with a name and type). By creating a directory and storage entries, the source language strings are stored "associatively" in a sense that they can be easily retrieved later based on the content. Such memory is therefore called associative memory. In a non-procedural language like NOPAL, source statements can be entered in any order, hence this capability of easy retrieval is very important to such a language processor.

The two RETRIEVE routines provide the user with easy access to the statement strings, based on some keys and by traversing the directory and storage entries.

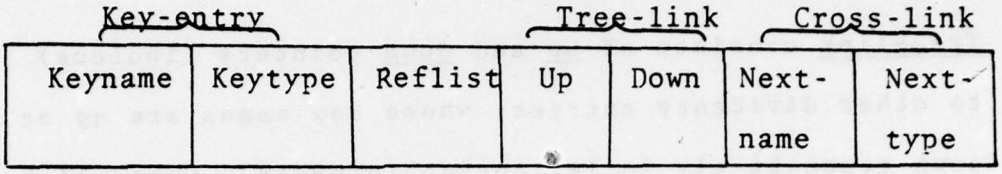
5.4.2 THE DIRECTORY AND STORAGE ENTRIES

The directory is a collection of directory entries. Each directory entry corresponds to a key which is composed of a name and type. Such an entry points to the last storage entry which contains the same key. A last-in-first-out (LIFO) linked-list is maintained from the most recent storage entry with that key to other storage entries containing the same key. The major advantage

of linking storage entries in LIFO order is that it is much faster to add a new storage entry to the top of the list. There is no need to traverse the list down to the bottom when adding a new entry, which is exactly the case if the list were maintained in first-in-first-out (FIFO) order. The directory itself is a binary tree structure, that is, each directory entry has a "up" pointer and a "down" pointer to other entries. Thus the first key entered in the directory becomes the root of the directory tree; the next key is entered "above" (linked via its "up" pointer) or "below" (linked via its "down" pointer) it is in the tree according to lexicographic order, and so on. This type of directory structure makes the modifications of the directory and the searching for keys more efficient.

Each directory entry has the forms as depicted in Figure 5.5(a). It consists of four fields (Key-entry, Reflist, Tree-link, and Cross-link), where Key-entry contains a key which is composed of two parts, keyname and keytype. Keyname is a variable string of up to 12 characters, serving as the name of the key. Keytype is an integer number denoting the type (or class)

(a) DIRECTORY ENTRY



(b) STORAGE ENTRY

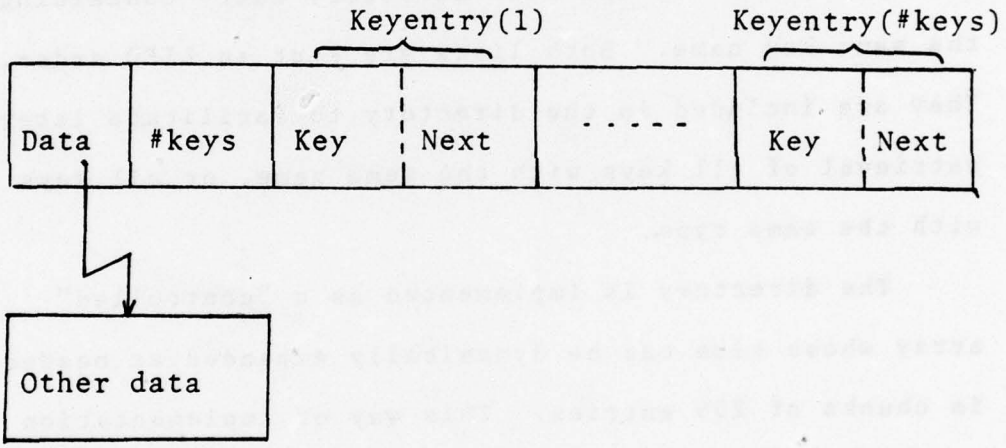


FIGURE 5.5 STRUCTURE OF THE DIRECTORY AND STORAGE ENTRY

of the key.

Ref-list is a pointer to the LIFO list of storage entries containing the key.

Tree-link consists of Up and Down pointers (indices) to other directory entries, whose key names are up or down respectively in lexicographic order. Cross-link consists of two fields (Nextname and Nexttype) of link pointers (indices). Nextname points to the next directory entry which contains the same key type. Nexttype points to the next directory entry containing the same key name. Both lists are kept in LIFO order. They are included in the directory to facilitate later retrieval of all keys with the same name, or all keys with the same type.

The directory is implemented as a "controlled" array whose size can be dynamically expanded as needed in chunks of 256 entries. This way of implementation speeds up dynamic allocation and processing of directory entries. Also this makes it possible that each directory entry can contain a variable length key name (in PL/I-F).

Each storage entry consists of two parts: (1) a list of keys (and link pointers) which are entered in the directory, and by which information can easily

be retrieved later; (2) other data from the source language of the source statement, although it is not used in the process of retrieval. The structure of each storage entry is shown in Figure 5.5(b) where Data is a pointer to the other data of the source statement.

#Keys is the number of keys in this storage entry. Key (i = 1 to #keys) is a pointer (index) to the directory entry which contains the key (name and type). Next (i = 1 to #keys) is a pointer to next storage entry which contains the same key, represented by key(i). Note that all such storage entries are threaded together in a LIFO list, which is pointed from the "Reflist" of the corresponding directory entry.

All types of storage entries including other data, of NOPAL statements are depicted in Appendix A.

Figure 5.6 illustrates an example of three storage entries and a directory consisting of only four entries that have been entered in that order. In the illustration, each key is designated by its key name and key type separated by a slash (/). There are four keys (B/1, B/2, C/1, A/3); hence four directory entries are created in that order, and B/1 is at the root of the directory tree. The picture shows the directory

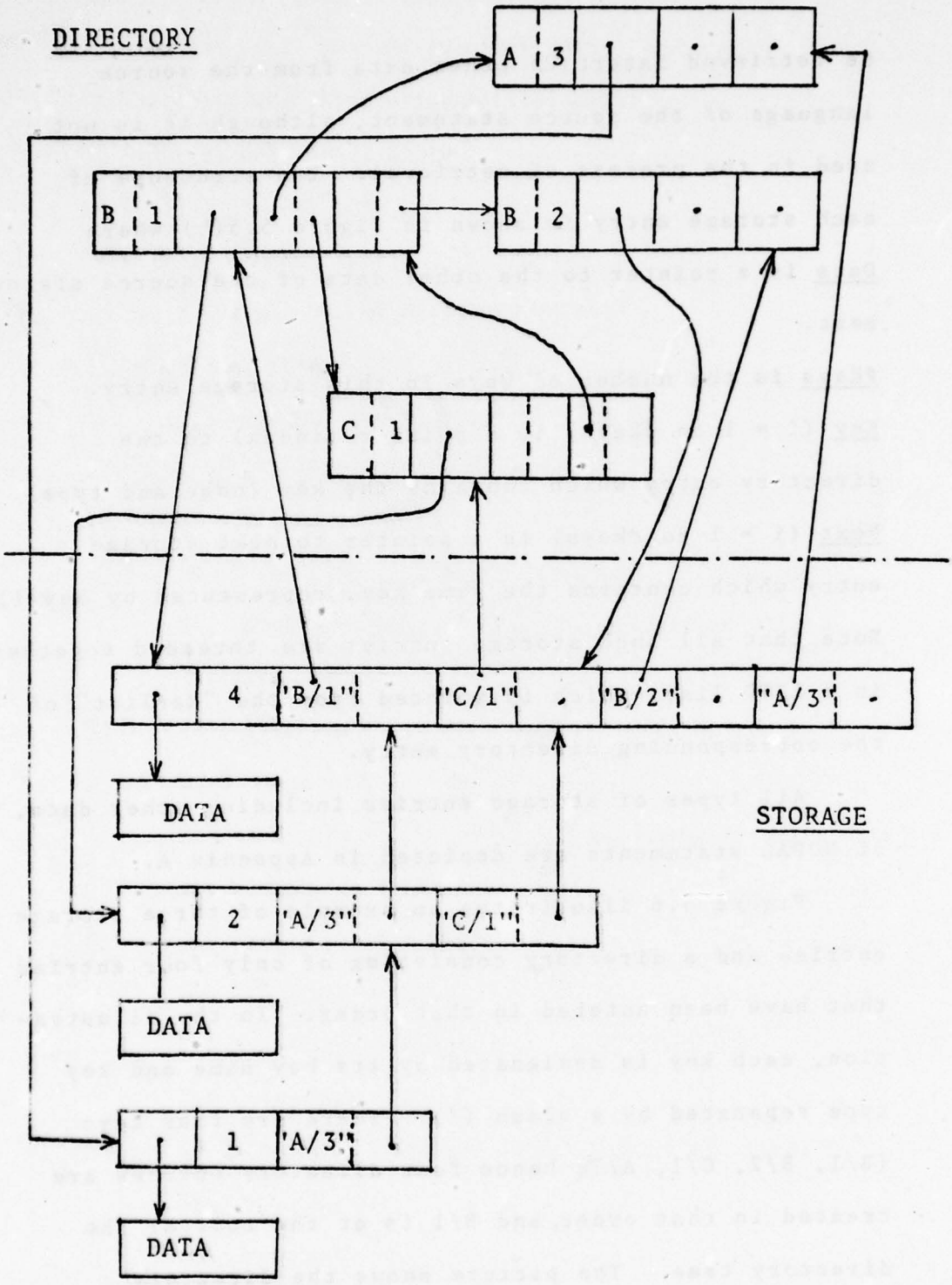


FIGURE 5.6 SAMPLE DIRECTORY AND STORAGE ENTRIES

and storage entries with all links after the three storage entries have been entered and created in that order.

5.4.3 The STORE Routine

The STORE(DP) routine has one formal parameter DP. But there are three common variables, N#, NAMES and TYPES, which are implicitly used in this routine. DP is a pointer to a previously created data from the source statement. N# is an integer denoting the number of keys to be stored in the storage entry and to be updated in the directory. NAMES is an array of key names, each is a variable-length string of up to 12 characters. TYPES is another integer array of key types. That is, the ith key has its name in NAMES(i), and type in TYPES(i). To be flexible, the two arrays are implemented as "controlled" storage that their size can be dynamically expanded in chunks of 64 elements when needed.

Each invocation of the STORE routine will create a storage entry containing N# keys and update the directory: the data structures have been depicted in Section 5.4.2.

Algorithm 5.1 shows the steps of this STORE routine. It obtains the keys (key names and types) from the two common areas (NAMES and TYPES) and creates a storage

ALGORITHM 5.1 STORE: SOURCE STRING

```
/*  One parameter, Dp: pointer to other data                    */
/*  Note the following three external variables:
      N# = no. of keys to be stored,
      NAMES(i) = ith key name;
      TYPES(i) = ith key type:                                   */
/*    #NODES = current no. of directory entries                  */
S1:   Allocate a storage entry;
      Set DATA pointer of the storage entry to Dp.
S2:   /* Check if the first time   */
      If #NODES > 0 then go to S3.
      S2.1: /*  First time: allocate directory   */
            Set MX#DIRS = 256;  /* Max. no. of entries  */
            Allocate DIRCTRY
      S2.2: /* Initialize First directory entry (i.e. root)  */
            Set# NODES = 1;
            Set #symbol = 1;      /* No. of key names */
            Set keyname(1) = NAMES(1);
            Set keytype(1) = TYPES(1);
            Set Ref_list  = Null;
            Set tree-link  = nil;
            Set Cross-link = nil.
      S2.3: /* Initialize TYPEhead:  list heads of all types */
            Set Typehead = 0;
            Set Typhead(TYPES(1)) = 1.
```

```
S3:   /*  Start processing each key  */
      For i = 1 to N#, perform steps S4 to S12.
      S4:  /*  Get keyname and type & initialize  */
           Set Knam = NAMES(i);
           Set Ktyp = TYPES(i);
           Set jn, jt = 1.  /* start at root */
      S5:  If Knam = keyname(jn), then go to S8;
           else if Knam < keyname(jn), then go to S7
      S6:  /* Knam > keyname(jn):  up */
           If Up(jn) = nil then
           Set Up(jn) = #NODES +1, and go to 7.1;
           otherwise set jn = Up(jn), and go to  S5.
      S7:  /*  Knam <keyname (jn):  down */
           If Down(jn) = nil, then
           set Down(jn) = #NODES + 1;
           otherwise set jn = Down(jn) and go to S5.
           S7.1:  Set jn = #NODES +1;
      S8:  /* Knam = keyname(jn):  check type */
           set jt = jn;
           While (jt ≠ nil), perform steps S8.1
           and S8.2.
           S8.1:  If ktyp = keytype(jt), then go to S12.
           S8.2:  Set jt = Nexttype (jt).
```

```
S9:  /*  Add new directory entry  */
     Set #NODES = #NODES + 1;
     If #NODES > MX#DIRS, then
          Call Expand_directory;
     Set jt = #nodes.
S10: /* Fill directory entry and update links */
     Set keyname (jt) = Knam;
     Set keytype (jt) = Ktyp;
     Set Nextname (jt) =  Typehead (ktype)
     Set Typehead (ktype) = jt.
S11: If jt = jn then go to S12.
     Set Nexttype(jt) = Nexttype(jn);
     Set Nexttype(jn) = jt
S12: /* Fill and link storage entry */
     Set Key(i) = jt;
     Set Next(i) = Ref-list(jt);
     Set Reflist(jt) to point to the storage entry.
S13: Return.
```

entry for them (step S1). If the routine is invoked for the first time, then the directory is allocated and initialized (steps S2 to S2.3). Otherwise, the algorithm searches the directory for a match of a key name (steps S3 to S7). If the keyname has been in the directory, then the algorithm proceeds to search for the keytype (steps S8 and S9). If either the keyname in the former search or the key type in the latter search is not found, then the key has not been entered in the directory, hence a new entry is created and filled (steps S9 and S11). Finally, the newly created storage entry is put on top of the "reference list" of the key (step S12). This process is repeated once for each key.

5.4.4 The RETRIEVE Routines

There are two procedures of this type: RETREVS and RETREVD. The former is used for retrieving desired storage entries; the latter for desired directory entries of a given key type or key name. The data structures depicted in Section 5.4.2 are used in these two routines.

The RETREVS (OPTION, RPTRS, NR, STMT_TYPE) procedure has four input parameters as indicated in parenthesis. In addition, it uses three common variables:

N#, NAMES and TYPES. The procedure finds all storage entries in which a conjunction of N# keys (specified in NAMES and/or TYPES) appears, and optionally checks other data associated with such storage entries. In other words, RETREVS retrieves all the storage entries with keys satisfying the conjunction and the data type. The pointers to the retrieved storage entries are returned in RPTRS, while the total number of such entries are returned in NR. Keys in conjunction may be negated, except the first one. A negation of a key is indicated by negating its corresponding key type. For example, if ith key is to be negated, then TYPES(i) should be negative. Finally, there are two options of specifying the keys: by directory locations (indices) of the keys if known, or by key names and types explicitly. The input parameters and common variables are summarized in the following.

OPTION is a one-bit flag indicating how the keys. If OPTION = 0, then the keys are specified by their directory locations in TYPES. Otherwise, the key names are in NAMES, and key types in TYPES.

RPTRS is an array of pointers to the storage entries retrieved.

NR contains the total number of such storage entries.

STMT_TYPE gives the statement type in the other data associated with the storage entries. If it is zero, then no check of data type is made. If positive, then the data type of each retrieved storage entry must be STMT_TYPE. Otherwise, STMT_TYPE is negative, the data type must not be STMT_TYPE.

N# contains the total number of keys.

NAMES is an array of key names, each is a variable-length string of up to 12 characters. It is not used if OPTION = 0.

TYPES is an integer array. The absolute value of TYPES(i) denotes the key type of the ith key, if OPTION = 1. Otherwise, it denotes the directory location of the ith key. If TYPES(i) is negative, then the ith key is negated in the conjunction.

To illustrate the usage of the RETREVS procedure, the following is an example of retrieving storage entries satisfying a conjunction of two keys: X of type CONJ#, and Y of type STIM# but negated (in NOPAL system, it means to retrieve all conjunctions named X which are not in the stimulus Y).

```
N# = 2;  NAMES(1)='X'; TYPES(1) = CONJ#;
         NAMES(2)='Y'; TYPES(2) = - STIM#;
CALL RETREVS('1'B, P, N, 0);
```

where P would contain a list of pointers to those retrieved storage entries, and N would be set to the number of such entries. Suppose the directory locations of these two keys have already been known as, say, K1 and K2, then the following calling sequence is equivalent to the one shown above:

```
N# = 2; TYPES(1) = K1; TYPES(2) = -K2;
CALL RETREVS ('0'B, P, N, 0);
```

Algorithm 5.2 shows the steps of RETREVS. If option 1 is used, then key names and types are explicitly specified and they are converted to the corresponding directory locations by searching the directory (steps S2 to S2.2). The algorithm then proceeds to search each storage entry containing the first (i.e. leading) key (steps S4-S5). If the data type associated with the storage entry does not match the desired statement type (STMT_TYPE), then the entry is skipped (steps S6-S6.3). If there are other keys in conjunction, then each of them must (or must not) be contained in the storage entry if it is non-negated (or negated) (steps S7-S7.5). If the storage entry turns out to be the

ALGORITHM 5.2 RETREVS: RETRIEVE STORAGE ENTRIES

/* There are four parameters:

Option = a one-bit flag indicating how the keys are provided for retrieval,

Rptrs = an array of pointers of the retrieval storage entries;

Nr = number such retrieved entries;

Stmt-type = statement type as additional condition for retrieval.

Also, the following 3 external variables are used:

N# = number keys specified for this retrieval;

Names = an array of key names, if option = 1;

Types = an array of key types, if option = 1;

or of directory locations of the keys, if option = 0. */

S1: If option = 0, then go to S3.

S2: /* Key names and types are explicitly specified */

For i = 1 to N#, perform steps S2.1 to S2.2.

S2.1: Search the directory for the key denoted by Names (i) and Types (i);

Let j be the location of the directory entry.

S2.2: Set Types (i) = j.

S3: /* Keys are specified by their locations (Types(*)) in the directory */

Set key#1 = Types(1);

Set Nr = 0;

Set Stoptr = Reflist (key#1).

S4: /* Trace each storage entry containing first key */

While (stoptr ≠ null), perform steps S5 to S9.

S5: Let k be the position of the key in the current storage entry.

S6: /* check statement type */

Let Data-type be the statement type stored in the "other data";

S6:1: If Stmt-type = 0 or Stmt type = Data type, then go to S7.

S6.2: If stmt type > 0 and Data type ≠ stmt type, then go to S9.

S6.3: If stmt type < 0 and Data type = Stmt type, then go to S9.

S7: /* Check other keys in conjunction, if any */

For i = 2 to N#, perform steps S7.1 to S7.5.

S7.1: Set Key # = Types(i); /* ith key */

S7.2: If Key # < 0, then go to S7.4.

S7.3: /* Key # > 0: in conjunction */

If Key # is not in the current storage entry, then go to S9;

else go to S7.5.

S7.4: /* Key # < 0: not in conjunction */

Set Key # = = Key #;

If Key # is in the storage entry, then go to S9.

S7.5: /* End of looping i */

S8: /* OK, the entry should be retrieved */

Set Nr = Nr +1;

Set RPTRS(Nr) = Stoptr;

S9: /* Get next storage entry */

Set Stoptr = Next(k).

S10: Return.

desired one, then its pointer is saved (step S8). Finally, the algorithm obtains the next storage entry in the "reference" list of the first key (step S9), and the process is repeated.

The other procedure RETREVD (OPTION, RNODES, NR) has three input parameters and two common variables NAMES and TYPES. If the one-bit OPTION is 0, then it retrieves all directory entries which contain the key type specified by TYPES(1). Otherwise, the algorithm retrieves all directory entries containing the keyname specified by NAMES(1). The locations of the retrieved directory entries are returned in RNDOES, and the total number in NR. Note that each key corresponds to one directory entry, as indicated by the structure of the directory (Section 5.4.2). For example, the following sequence of PL/I code retrieves all directory entries of key type TEST# (in NOPAL system, it is equivalent to obtaining the directory locations of all test module names):

```
TYPES(1) = TEST#; CALL RETREVD('0'B, R,N);
```

where R would contain those directory locations, and N would contain the total number of them.

Algorithm 5.3 outlines the steps of the procedure RETREVD. If option 0 is used, then the key type is

```
ALGORITHM 5.3  RETREVD:  RETRIEVE DIRECTORY ENTRIES

     /*  There are three parameters:
         Option = a one-bit flag indicating mode of
         retrieval,
         Rnodes = an array of directory entries retrieved;
         Nr = number of such entries.
         In addition, the following two external variables
         are used:
         Names = an array of key name, if option = 1;
         Types = an array of key types, if option = 0 */
S1:  If option = 1, then go to S5.
S2:  /* Option = 0; retrieves all div.entries with same type
     */
     /* Get the keytype from types(1)  */
     Set type = Types(1);
     Set k = Typehead (type).
S3:  While k ≠ nil, perform steps 3.1 to 3.2.
     S3.1:  Set Nr = Nr + 1;
            Set Rnodes(Nr) = k
     S3.2:  Set k = Nextname (k).
S4:  Go to S8
S5:  /* Option = 1, retrieve all dir. entries with same
     name  */
     /* Get the key name from Names(1)   */
     Set kname = Names(1);
     Set k = 1;
```

```
S6:  /* Search key name */
     While k ≠ nil, perform step S6.1.
     S6.1:  If kname > Keyname(k) then
            set kn = Up(k);
            else if kname  <Keyname(k) then
             set kn = Down(k);
            else go to S7.  /* name found */
S7:  /*  Get all entries with the same name */
     While k = nil, perform steps S7.1 to S7.2.
     S7.1:  Set Nr = Nr + 1;
            Set Rnodes (Nr) = k.
     S7.2:  Set k = Nexttype (k).
S8:  Return.
```

given in TYPES(1) and the algorithm gets all the directory entries with the same key type via a link field, NEXTNAME(steps S2-S4). Otherwise, the key name is specified in NAMES(1), and the directory is searched for the first entry containing the key name (steps S5-S6.1). Then the algorithm obtains all the directory entries containing the same key name via another link field, NEXTTYPE (steps S7-S7.2).

5.5 NOPAL SPECIFICATION REPORTS

There are two NOPAL specification reports which are available to the user at his discretion. The Source specification report is a by-product of the syntax analysis phase. The reformatted specification report is produced from the stored NOPAL statements in the simulated associative memory. These two reports are briefly discussed in the next sub-sections respectively.

5.5.1 Source Specification and Syntax Error Reports

The source specification report is the image of the NOPAL specification provided by the user, except that each statement is prefixed by a corresponding statement number generated by the Processor. These statement numbers are referenced in the syntax error report, the cross-reference-attribute report, and the cross-

reference error report (the last two reports are discussed in Section 5.6). Therefore this report is useful to the user during the debugging stage. As shown in Figure 5.3, the report is generated by the lexical analyzer (LEX) as a by-product. This report may optionally be suppressed by giving a run-time parameter (NOSAPLIST). A sample source specification report is given in Figure 3.2.

The Syntax error report is another document which lists all syntax error or warning messages detected by the Processor during the syntax analysis phase. As indicated in Figure 5.3 this report is generated by a collection of error message routines (see Section 5.3.2). If an error is encountered in a statement, then the corresponding error code and statement number will appear in this report. The error codes are enumerated in Table 5.4. In the case of warning messages, they are usually informatory.

5.5.2 Reformatted Specification Report

A specification report, reformatted and reorganized for better readability, is produced by a program module (SOURCE2) whose input is the whole collection of NOPAL statements stored in the simulated associative memory. Basically this reformatted specification report lists

the three major sections (test-modules, UUT, and ATE) of the NOPAL specification in that order, with proper headings and indentation. All default information is provided in this report. Thus, this is a complete, purely non-procedural NOPAL specification and is acceptable to the NOPAL Processor. This report is useful to the user, particularly when his source specification is not well-organized.

The SOURCE2 routine produces the reformatted specification report by traversing the directory and storage entries using RETRIEVE subsystem. This report, can be suppressed by giving a run-time parameter (NOSOURCE2).

Figure 5.7 shows a portion of a typical reformatted specification report.

5.6 Cross Reference Reports

This section presents the subsystem which produces a set of six cross reference reports as summarized in Table 5.1. As indicated in Figure 5.3, these reports are generated by a program module XREF (in fact, XREF1 and XREF2) whose input is the stored NOPAL specification. Section 5.6.1 describes the cross-reference and attribute report produced by XREF1. Section 5.6.2 presents the other cross-reference

```
/* REFORMATTED SPECIFICATION REPORT, FILE: SOURCE2 */

/*****************************************************/
/*                                                   */
/*  NOPAL TEST SPECIFICATION FOR MINIRADIOSET        */
/*                                                   */
/*****************************************************/

NOPAL SPECIFICATION MINIRADIOSET;

/*****************************************************/
/*                                                   */
/*  TEST MODULES:     6                              */
/*                                                   */
/*****************************************************/


TEST DC_INPUT;

 /* NULL STIMULI */

 MEASUREMENT $M_DC_INPUT(DC_INPUT);

    CONJUNCTION $M_W0001($M_DC_INPUT):
       (<J24_B, J24_C>  = CONST_R(MRES OHM ))
                TARGET: MRES;

    ASSERTION $M_W0002($M_DC_INPUT):
       MRES >  100
                SOURCE: MRES;

 LOGIC $LOGIC001C(DC_INPUT): |¬D2, *D3;

    DIAGNOSIS D2:
       OPERATOR MESSAGE:
          AFFECTED COMPONENTS=INPUT_SHORT,
          TYPE=#4;

    DIAGNOSIS D3:
       OPERATOR MESSAGE:
          OTHER PARAMETERS=(MRES, ' OHMS'),
          TYPE=D;
```

FIGURE 5.7 EXAMPLE OF REFORMATTED SPECIFICATION REPORT

reports generated by XREF2.

5.6.1 Cross Reference and Attribute Report (XREF-ATTR)

The XREF-ATTR report is a useful product of the syntax and statement analysis phase. It is produced by a cross-reference routine (XREF1) by inputting the NOPAL statements stored in the simulated associative memory. This report provides an alphabetical listing of all the names (as identifiers or labels) defined by the user in the submitted NOPAL specification. For each name, the XREF-ATTR report gives the statement number of the statement which defines the entity, the statement numbers of the statements which reference the name, and a list of attributes regarding the name. Thus, this report is useful to the user during the debugging stage.

Figure 5.8 shows an example of a typical XREF-ATTR report.

The printing of this report can be suppressed by giving a run-time parameter (NOXREF1).

The XREF1 routine produces this report by traversing the directory and by invoking the RETRIEVE routine to obtain the corresponding references. Since the directory

CROSS REFERENCE AND ATTRIBUTES REPORT

NAME	DEF NO.	ATTRIBUTES AND REFERENCES
#15	56	MESSAGE LABEL
		31
#17	57	MESSAGE LABEL
		33
#18	58	MESSAGE LABEL
		22 51
#4	53	MESSAGE LABEL
		7
#5	54	MESSAGE LABEL
		41
#6	55	MESSAGE LABEL
		14 42
AMPL	9	TEST LABEL
		10 12
AMPL_TOL	78	ATE-FUNCTION ID, F
		14 68
ATE_J24B	81	ATE-POINT ID
AUDIO_10MW	69	COMPONENT ID, WITH FAILURE-FUNCTION: REF_VOLT
		33
AUDIO_10MW	70	COMPONENT ID, WITH FAILURE-FUNCTION: DISTORT
		51
AUDIO_2W	71	COMPONENT ID, WITH FAILURE-FUNCTION: DISTORT
		22
A1	28	ASSERTION LABEL
A2	29	ASSERTION LABEL
CONST_R	73	ATE-FUNCTION ID, M
		4
CONST_S	72	ATE-FUNCTION ID, S
		36 25 17 45
D	52	MESSAGE LABEL
		8 13 32 43 50
DC_INPUT	2	TEST LABEL
		3 6
DCV	35	STIMULUS LABEL
		36 25
DCV_AMS	24	STIMULUS LABEL
		25 17 45
DISLPAY	52	SYNONYM OF MESSAGE LABEL: D
DISTORT	80	ATE-FUNCTION ID, F
		22 51 70 71
DISTORT_VOLT	23	TEST LABEL
		24 26 30
DISTORT_10MW	44	TEST LABEL
		45 46 49
DISTORT_2W	15	TEST LABEL
		16 18 21
DISTORTION	75	ATE-FUNCTION ID, M
		19 47
D2	7	DIAGNOSIS LABEL
		6
D3	8	DIAGNOSIS LABEL
		6
D4	41	DIAGNOSIS LABEL
		40
D5	42	DIAGNOSIS LABEL
		40

FIGURE 5.8 EXAMPLE OF XREF-ATTR REPORT

is itself a binary-tree structure, an alphabetic ordering of names is easily achieved by an in-order traversal of the directory (i.e., left subtree first, then the node, finally the right subtree).

Any errors or warnings which are detected during this phase of cross-referencing are displayed in a separate XREF1 error report.

During the process of generating this report, three other minor tasks are also accomplished. First, all the stimulus/measurement waveform conjunction back references are resolved before the production of the XREF-ATTR report is actually begun. Second, for each variable, its scope (global or local) is determined. Furthermore, an occurrence of a variable in a conjunction or an assertion is considered as SOURCE by default, hence it is put in the corresponding source-variable list unless it is otherwise declared as TARGET. The detailed steps are provided in Algorithm 6.1 of Chapter 6. Last, to facilitate later phases of processing, the stimulus, measurement, and logic-diagnosis list of each test module are explicitly linked to the test module. Similarly, the conjunction and assertions are explicitly linked to their corresponding parent stimuli or measurement.

5.6.2 Other Cross Reference Reports: DIAG-TEST, MESS-DIAG-TEST, COMP-DIAG-TEST, UUT.PT-TEST-ATE.PT and FUNC-TEST

Available to the user are five additional cross-reference reports which are generated from the stored NOPAL statements by a program module (XREF2). Basically, they are reports in which the test modules are cross-referenced with the diagnosis, the messages, the affected components, the UUT and ATE connecting points, or the ATE functions (as enumerated in Table 5.1). These reports provide better man-machine interface and give the user a clear picture of interactions among various components of his test specification. They may be entirely suppressed by giving a run-time parameter (NOXREF2).

In the DIAG-TEST report diagnoses are cross-referenced with the test modules. For each diagnosis, this report lists the names of the test modules which ever references the diagnosis.

The MESS-DIAG-TEST report provides a listing of all the message names, together with the corresponding diagnoses and test modules which refer to them.

The COMP-DIAG-TEST report lists all the affected components (i.e., component failures), with all the referencing diagnoses and test modules.

The UUT.PT-TEST-ATE.PT report lists all the UUT connecting points. For each UUT connecting point, the report provides all the test modules referencing it, with the stimulus or measurement sections properly suffixed. Also provided in the report is a list of ATE interconnecting points which are connected directly, or indirectly through ATE-UUT interface, with the given UUT connecting point.

The last cross reference report, FUNC-TEST report provides a list of ATE functions. For each function, the report lists all of its referencing test modules, with stimuli or measurements properly suffixed.

Figure 5.9 through 5.11 show an example of these five cross-reference reports.

These cross-reference reports should be useful to the user as a debugging aid or for the purpose of better understanding his own NOPAL test specification.

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- DIAGNOSES <=> TEST-MODULES

DIAGNOSIS	TEST-MODULES
D2	DC_INPUT
D3	DC_INPUT
D7	AMPL
D8	AMPL
27	DISTORT_2W, DISTORT_10MW
30	DISTORT_2W
24	DISTORT_VOLT
25	DISTORT_VOLT
26	DISTORT_VOLT
D4	FREQ
D5	FREQ
D6	FREQ
28	DISTORT_10MW

FIGURE 5.9 EXAMPLE OF DIAG-TEST REPORT

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- MESSAGES <=> DIAGNOSES <=> TESTS

MESSAGE	DIAGNOSES	TEST-MODULES
#4	D2	DC_INPUT
D/DISLPAY	D3, D4, 25, D6, 27	DC_INPUT, AMPL, DISTORT_VOLT, FREQ, DISTORT_2W, DISTORT_10MW
#6	D8, D5	AMPL, FREQ
#18	5C, 28	DISTORT_2W, DISTORT_10MW
#15	24	DISTORT_VOLT
#17	28	DISTORT_VOLT
#5	D4	FREQ

(a) MESS-DIAG-TEST REPORT

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- AFFECTED-COMPONENTS <=> DIAGNOSES <=> TESTS

AFFECTED-COMPONENT	DIAGNOSES	TEST-MODULES
1: INPUT_SHORT	D2	DC_INPUT
2: FREQ_TOL(STD_5MHZ_FREI	D5	FREQ
3: AMPL_TOL(STD_5MHZ_FREI	D8	AMPL
6: REF_VOLT(AUDIO_10MW)	26	DISTORT_VOLT
7: DISTORT(AUDIO_10MW)	2A	DISTORT_10MW
00600: DISTORT(AUDIO_2W)	3C	DISTORT_2W

(b) COMP-DIAG-TEST REPORT

Figure 5.10

Examples of MESS-DIAG-TEST and COMP-DIAG-TEST Reports

(a) UUT.PT-TEST-ATE.PT REPORT

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- UUT-CONNECTING-POINTS <=> TEST-MODULES <=> ATE-CONNECTING-POINTS

UUT-CONNECTING-POINT	TEST-MODULES(S/M)	ATE-CONNECTING-POINTS
J24_B/XJ24_B	DC_INPUT(M), FREQ(S), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	ATE_J24B
J24_C/GND	DC_INPUT(M), DISTORT_2W(M), DISTORT_VOLT(M), FREQ(S), FREQ(M), DISTORT_10MW(M), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	
J19_L	DISTORT_2W(M)	
J16	DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	
J19_A	DISTORT_VOLT(M), DISTORT_10MW(M)	
J22	FREQ(M)	
J19_B/GND	DISTORT_2W(M), DISTORT_VOLT(M), FREQ(S), FREQ(M), DISTORT_10MW(M), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	

(b) FUNC-TEST REPORT

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- ATE-FUNCTIONS <=> TEST-MODULES

ATE-FUNCTION,TYPE	TEST-MODULES(S/M)
CONST_B, M	DC_INPUT(M)
DISTORTION, M	DISTORT_2W(M), DISTORT_10MW(M)
SIGNAL_AM/SAM, S	DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)
SINE_O/SINE_DELAY, M	DISTORT_VOLT(M), FREQ(M)
CONST_S, S	FREQ(S), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)

Figure 5.11

Examples of UUT.PT-TEST-ATE.PT and FUNC-TEST Reports

CHAPTER 6

SPECIFICATION ANALYSIS AND SEQUENCE DETERMINATION

6.1 INTRODUCTION AND BACKGROUND:

This phase of the NOPAL processor deals with the analysis of NOPAL specification, automatic program design, and determination of sequence based on an application of graph theory. The analysis of information relationships and automatic sequencing by means of graphs have been successfully used in the recent MODEL system [RIN 76]. This section presents the background and terminology involved in this phase. It also describes the graphs, matrices, and other data structures that are generated from a NOPAL specification.

In order to explain the algorithms and data structures used, the sample NOPAL specification MINI-RADIOSET, presented in Figure 3.2 will be frequently referred to in the subsequent discussions.

In a NOPAL specification, each entity (e.g. test, diagnosis, conjunction, assertion, and variable) is given a symbolic name which is either provided by the user or generated by the Processor. In this phase each name is related to others in one of the several ways.

For example, a data determinacy relationship exists between a test module having a global TARGET variable and the variable, also between a global variable and a test module referring to the variable as SOURCE.

In all these relationships, referred to as precedence relationships, the former in a sense must precede the latter at execution time of the object test program and is said to be a predecessor of the latter, while the latter is said to be a successor of the former. All types of precedence relationships that exist among test modules are summarized in Table 6.1. As for the precedence relationships internal to a given test module, there are merely two types, data determinacy and waveform setup. These are further explained later. The types of precedence relationships dictate the following: 1) how the conjunction and assertions are analyzed and sequenced internally, 2) how the test modules are analyzed and sequenced externally, and 3) how the object program is generated. For instance, a global variable must be evaluated first in the predecessor test module before the variable can be used in another successor test module. Such precedence information is conveyed in a "directed graph", as described below, one for the entire aggregate of test modules and one for each test module.

TABLE 6.1 INTER-TEST MODULE PRECEDENCE RELATIONSHIPS

Precedence			Relationship selection rule		Run-time condition	Explanation
Type	Priority	Strategy name	PREDECESSOR	SUCCESSOR		
1	1	Data determinacy	(a)Test module having global TARGET variable X or (b)Global variable X	(a)Variable X or (b)Test module using X as SOURCE		Global variable is evaluated in predecessor or referenced in successor
2	1	Interactiveness	Diagnosis D	Test module connected with D by "after"(A)	D's operator response Y	Test module is started after response Y
3	1		Diagnosis D	Test module connected with D by "after-not"(A¬)	D's operator response N	Test module is started after response N
4	1	Component protection	Diagnosis D	Test module with an affected component protected by one of D's	D is not selected	Failure of critical component prohibiting testing other components
5	2	Fault isolation	Diagnosis D whose affected components are in disjunction	Test module whose affected components set is a subset of D's	D is selected	If D asserts more generic failures, more specific tests are conducted
6	2		Diagnosis D whose affected components are in conjuction	Test module whose affected components set is a subset of D's	D is not selected	If D isolates some faults, skip tests for subset of the same faults
9	3	Stimuli applica-	Test module whose most frequent stimuli	Test module whose most frequent sti-		Once a stimuli is setup, as many

TABLE 6.1 (continued)

Precedence			Relationship selection rule		Run-time condition	Explanation
Type	Priority	Strategy name	PREDECESSOR	SUCCESSOR		
		tion	triplet is more frequent	muli triplet is less frequent		tests as possible are performed
10	4	Failure likelihood	Test module whose smallest failure index of affected components is smaller	Test module whose smallest failure index of affected components is larger		Tests whose components are more likely to fail are performed first
11	1	Logical operator	Test module T	Diagnosis selected in T by operator "don't-care"(*)		Diagnoses are posted after the test module concludes. Types 11 to 15 may be combined into a type, but they are separated to speed up later processing.
12	1		Test module T	Diagnosis selected in T by operator "or"(|)		
13	1		Test module T	Diagnosis selected in T by operator "or-not"(|¬)		
14	1		Test module T	Diagnosis selected in T by operator "and"(&)		
15	1		Test module T	Diagnosis selected in T by operator "and-not"(&¬)		

Formally, a directed graph [AHO 74, DEO 74] (or a digraph) G = < N, E > consists of a finite, non-empty set of nodes (or vertices) N and a set of edges (or arcs) E where each edge is an ordered pair (t,h) of nodes; t is called the tail and h the head of the edge (t,h). Node h is said to be adjacent to node t, while edge (t,h) is said to be from t to h.

A weighted directed graph is a directed graph in which each edge (t,h) from node t to node h is associated with one of a set of types of relationships.

Weighted directed graphs are used to represent all the different types of precedence relationships derived from the NOPAL statements. As an example, the weighted digraph shown in Figure 6.1 corresponds to the inter-test-module relationships of the example of Figure 3.2. In this graph each node represents the name of one of the entities in the NOPAL specifications: test modules, diagnoses, and (global) variables. Note that each node may have 0, 1, or more edges emanating from it to successor nodes.

Although a pictorial representation of a graph is very convenient for visual study, there are other representations which are better for computer processing [DEO 74]. One such representation called adjacency matrix is used in the analysis phase of the NOPAL processor.

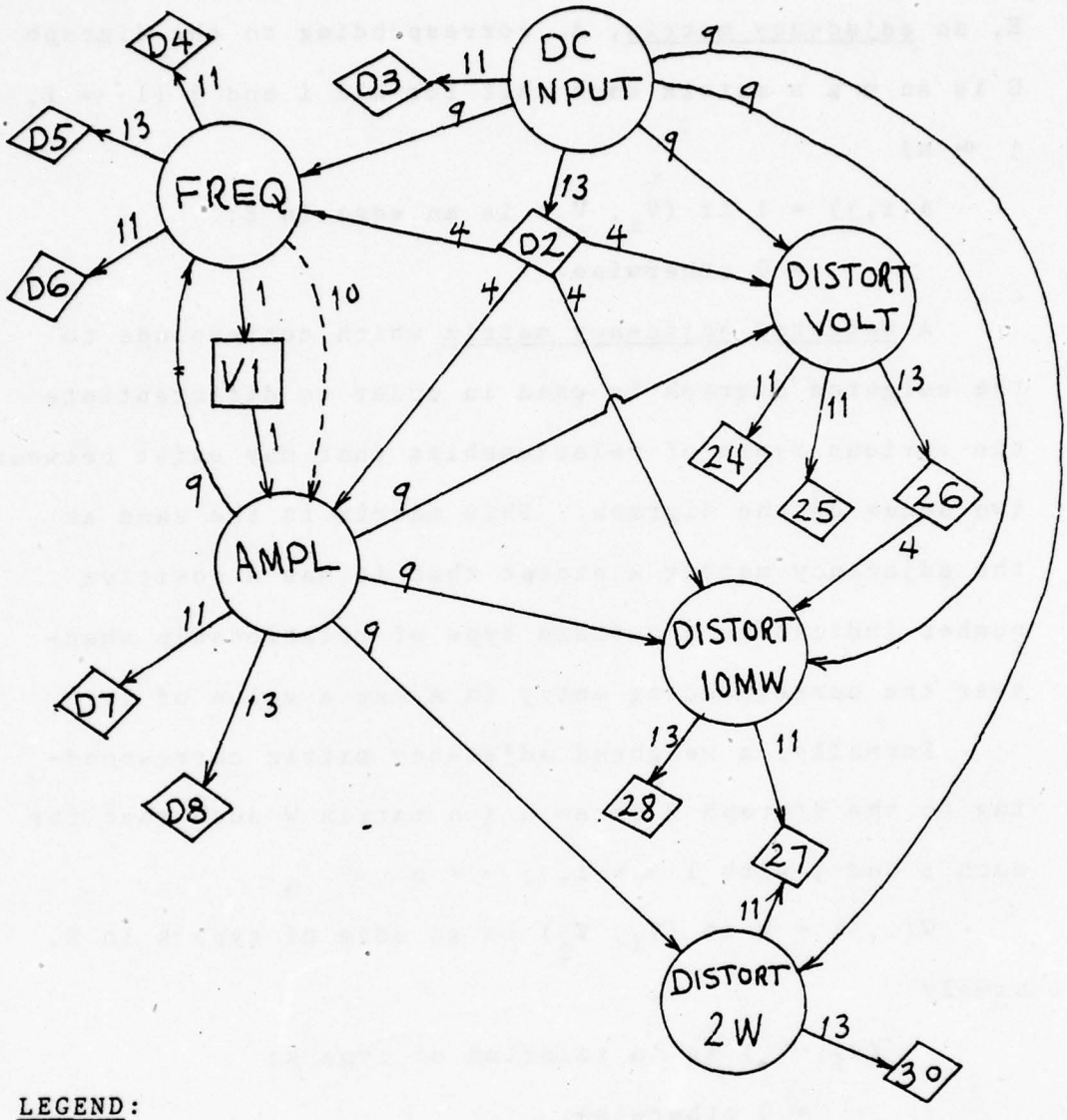


Figure 6.1 Digraph for NOPAL Specification MINIRADIOSET

Given a digrpah G = < N, E > consisting of a set of n nodes N = { V_1, V_2, ..., V_n } and a set of edges E, an <u>adjacency matrix</u>, A, corresponding to the digraph G is an n x n matrix such that for all i and j (1 <= i, j <= n)

A(i,j) = 1 if (V_i, V_j) is an edge in E;

= 0 otherwise.

A <u>weighted adjacency matrix</u> which corresponds to the weighted digraph is used in order to differentiate the various types of relationships that may exist between two nodes of the digraph. This matrix is the same as the adjacency matrix A except that it has a positive number indicating a certain type of relationship whenever the corresponding entry in A has a value of 1.

Formally, a weighted adjacency matrix corresponding to the digraph G is an n x n matrix W such that for each i and j with 1 < = i, j < = n

W(i,j) = k if (V_i, V_j) is an edge of type k in E, namely

(V_i, V_j) is in relation of type k;

= 0 otherwise.

The weighted adjacency matrix for a NOPAL specification is used extensively in the phases of analysis, sequencing, and code generation. Such a matrix corresponding to the MINIRADIOSET example of Figure 3.2 (hence the digraph of Figure 6.1) is shown in Figure 6.2. The node numbers to the left of the node names are assigned by the

Processor. Entries (i,j) in the matrix are either 0, indicating no relationship exists between the node i (at ith row) and the node j (at jth column), or a positive number corresponding to the type of precedence relationship between node i and node j. These type numbers correspond to the precedence types listed in Table 6.1. To further illustrate the precedence relationships, those precedence types and relationships which are extracted from the sample specification MINIRADIOSET are enumerated in Table 6.2.

All the global precedence information is conveyed by a weighted adjacency matrix for the whole NOPAL specification. Likewise, all the local precedence information in each test module is also represented by a weighted adjacency matrix. Such precedence information is entered into the matrices and analyzed in subsequent sections.

As illustrated in Figure 6.3, there are two major subphases of analysis and sequencing: intra-test-module and inter-test-module. The former concentrates internally on the waveform conjunctions, assertions, and diagnoses of a given test module. The latter focuses externally on the whole collection of test modules in a given NOPAL specification, considering each test module as an integral unit.

Section 6.2 gives an overview of the subphases common to both the intra- and inter-test module analysis and sequencing. Section 6.3 presents in detail all the subphases of intra-test-module analysis and sequence determination. Section 6.4 discusses various subphases of inter-test-module analysis and sequencing.

			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	DC_INPUT	TEST	0	0	9	9	9	9	13	11	0	0	0	0	0	0	0	0	0	0	0	0
2	AMPL	TEST	0	0	9	9	9	9	0	0	0	0	0	11	13	0	0	0	0	0	0	0
3	DISTORT_2W	TEST	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	13	0
4	DISTORT_VOLT	TEST	0	0	0	0	0	0	0	0	0	0	0	0	0	11	12	13	0	0	0	0
5	FREQ	TEST	0	10	0	0	0	0	0	0	11	13	11	0	0	0	0	0	0	0	0	1
6	DISTORT_10MW	TEST	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	13	0	0
7	D2	DIAG	0	4	0	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	D3	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	D4	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	D5	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	D6	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	D7	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	D8	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	24	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	25	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	26	DIAG	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	27	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	28	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	30	DIAG	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	V1	VAR	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

FIGURE 6.2 WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION OF MINIRADIOSET

TABLE 6.2 ILLUSTRATION OF PRECEDENCE RELATIONSHIPS FOR MATRIX OF FIGURE 6.2

Precedence type/relationship		Explanation
1	Data determinacy	Test FREQ(node 5) generates a TARGET variable V1(node 20), which is in turn used as SOURCE in test AMPL(node 2). Thus enties (5,20) and (20,2) have a 1.
4	Component protection	(a)Component INPUT_SHORT in diagnosis D2(node 7) protects FREQ_TOL(STD_5MHZ_FREQ), AMPL_TOL(STD_5MHZ_FREQ) REF_VOLT(AUDIO_10MW), and DISTORT(AUDIO_10MW) in diagnoses D5, D8, 26, and 28 respectively. The last four diagnoses are in tests FREQ, AMPL, DISTORT_VOLT, and DISTORT_ 10MW(nodes 5,2,4, and 6) respectively. Hence entries (7,5), (7,2), (7,4), and (7,6) have a 4. (b)DISTORT(AUDIO_10MW) is also protected by REF_VOLT(AUDIO_10MW), which is in turn in diagnosis 26(node 16) hence entry(16,6) has a 4.
9	Stimuli application	After all stimuli triplets have been counted and indexed(see Section 6.4.2.5), the most frequent stimuli triplet is J24_B,GND = CONST_S(27.5VOLT), which appears in the last 4 tests. The first two tests have no stimuli, hence entries (1,3), (1,4), (1,5), (1,6), (2,3), (2,4), (2,5), (2,6) have a 9.
10	Failure likelihood	FREQ_TOL(STD_5MHZ_FREQ) in test FREQ(node 5) and AMPL_ TOL(STD_5MHZ_FREQ) in test AMPL(node 2) have failure indices 1 and 2 respectively. Hence entry (5,2) is 10
11	Logical operator(*)	Diagnosis D3 in test DC_INPUT; D7 in AMPL; 27 in DISTORT_2W; 24 and 25 in DISTORT_VOLT; 27 in DISTORT_ 10MW; D4 and D6 in FREQ.
13	Logical operator(|¬)	Diagnosis D2 is in test DC_INPUT; D8 in AMPL; 30 in DISTORT_2W; 26 in DISTORT_VOLT; 28 in DISTORT_10MW; and D5 in FREQ.

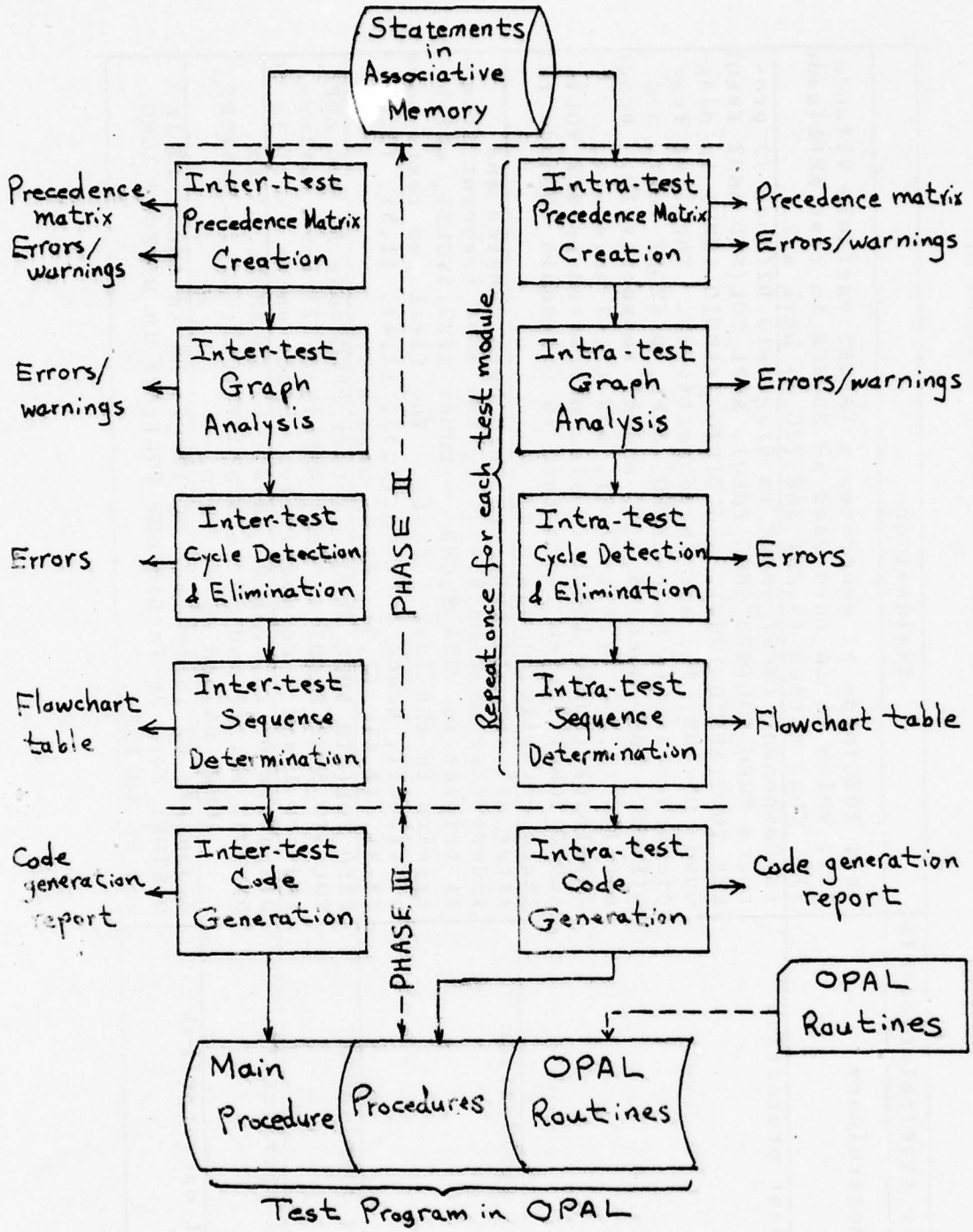


FIGURE 6.3 FLOWCHART FOR PHASES II AND III OF NOPAL PROCESSOR

6.2 OVERVIEW OF SUBPHASES IN GRAPH ANALYSIS AND SEQUENCING

A weighted digraph of the NOPAL specification as represented by a weighted adjacency matrix is a vehicle used by the NOPAL Processor to sequence operations and to detect errors in the specification.

Before and during the process of gathering and entering precedence relationships in the weighted adjacency matrix, some logic errors in the specification may be detected, and appropriate messages sent to the user. Further error analysis occurs after the matrix has been constructed. Table 6.3 summarizes the error/warning messages which can possibly be produced by the Processor during the phase of graph creation and analysis (after the syntax analysis phase). The reference numbers in the first column will be referred to occassionally during the subsequent discussions.

Variables that are global (or local) to a test module are involved in determining one type of precedence relation (data determinacy) in the phase of inter-test-module (or intra-test-module) analysis and sequencing. Therefore, the scope (global or local to a test module) of each variable in the whole NOPAL specification must be determined before the actual analysis and sequencing of test modules begin (internally and externally).

Basically if a variable X has ever been defined as TARGET alone, or used as SOURCE alone in any test module of the NOPAL specification, then X is considered as a global variable (i.e., of scope global). Otherwise, X is local (i.e., of scope local) to each respective test module where it has been both defined as TARGET and used as SOURCE. Algorithm 6.1 determines the scope (local or global) of every variable in the NOPAL specification. It also designates as SOURCE every variable which has never been explicitly declared as TARGET or SOURCE by the user. This relieves the user of the burden of declaring SOURCE variables explicitly. For the sake of efficiency, this algorithm is imbedded in the XREF1 routine (for cross-reference and attributes in Chapter 5) in current implementation.

Table 6.4 summarizes the steps involved in the creation and analysis of the weighted adjacency matrix representation of a digraph and in the determination of sequence, commonly applicable to both the internal and external analysis and sequencing of test modules. Detailed sub-phases in intra-test-module and inter-test-module analysis and sequencing are presented in Sections 6.3 and 6.4, respectively.

TABLE 6.3

ERROR/WARNING MESSAGES (XREF/ANALYSIS)

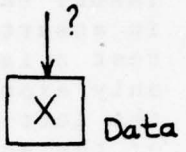
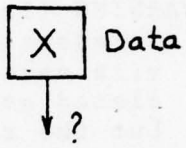
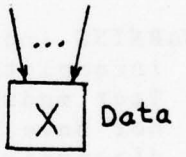
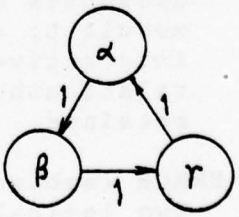
Ref#	Message	Issued by	Brief explanation/Example
1	ERROR (incompleteness): Variable X is used as SOURCE in ...; but its target definition never given elsewhere.	EXTSEQ	
2	WARNING (possible incompleteness): Variable X is defined as TARGET in ...; but never used elsewhere	EXTSEQ	
3	WARNING (possible ambiguity): Variable X is defined as TARGET more than once in ...; they must be under mutually exclusive condition.	INTSEQ & EXTSEQ	
4	ERROR (ambiguity): In assertion x of test y, there are two or more TARGET variables: ...	INTSEQ	Two or more TARGET variables in an assertion
5	ERROR (inconsistency): The following items are circularly related with procedence priority 1: α,β, r, ...	CYCLES	

TABLE 6.3

ERROR/WARNING MESSAGES(XREF/ANALYSIS)(CONTINUED)

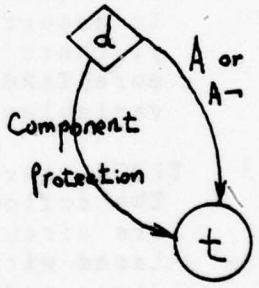
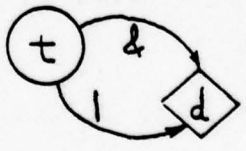
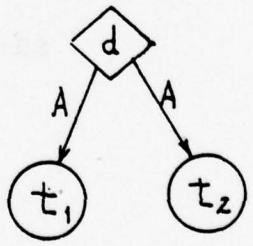
Ref#	Message	Issued by:	Brief explanation/Example
6	ERROR (ambiguity): TARGET variable x in assertion y of test z is <u>not</u> the only expression at the left-hand side of the equal sign ("=")	INTSEQ	W = X + 1 TARGET: X; <u>or</u> X - 1 = W TARGET: X;
7	WARNING (Inconsistency): In assertion x of test y, a variable is declared as target, but the relation operator is not an equal sign ("="). Replaced by an equal sign.	INTSEQ	V > W + 3 TARGET: V;
8	WARNING (possible incompleteness): Test module X does not have any diagnosis	EXTSEQ	A test has null logic-diag. list
9	WARNING (possible inconsistency): Both interactiveness and component protection relationships exist between diagnosis d and test module t; only the interactiveness relationship is retained	EXTSEQ	
10	ERROR (ambiguity): Two logical operators x,y connect test module t with diagnosis d.	EXTSEQ	

TABLE 6.3

ERROR/WARNING MESSAGES(XREF/ANALYSIS) (CONTINUED)

Ref#	Message	Issued by:	Brief explanation/Example
11	WARNING (possible ambiguity): X is defined/used as a local variable in tests...; and also in diagnoses ...	XREF1	Local variable X used in two diagnoses of two different tests.
12	WARNING (apparent inconsistency): In stmnt xxxx, X multiply defined. The stmnt deleted.	XREF1	Multiple statement definition e.g. TEST X; ⋮ TEST X;
13	WARNING (possible incompleteness): In stmnt xxxx, Diagnosis X never referenced.	XREF1	Diagnosis X was defined, but never used.
14	ERROR (inconsistency): In stmnt a,b,c,... conjunction back references form a loop.	XREF1	Circular stimuli conjunction back references.
15	ERROR (inconsistency): Two or more tests:... come after the diagnosis d via logical operator after ('A') or after-not ('A¬')	EXTSEQ	

ALGORITHM 6.1 DETERMINE SCOPES OF VARIABLES AND IDENTIFY SOURCE VARIABLES

```
S0:  For each variable X in the NOPAL specification,
     perform steps S1 through S44.
     S1:  Get all #SE storage entries of X;
          Set #DEF, #REF, ND = 0.
     S2:  For i = 1 to #SE, perform steps S3 to S10.
          S3:  Set DEF(i), DONE(i) = 0; /* false */
          S4:  If ith statement type is not diagnosis
               then go to S7:
          S5:  /* Setup "used" list of referencing logic
               entries.
               Not relevant here */
          S6:  If X is not an operator response variable,
               then go to S10; else go to S9.
          S7:  /* Set up test-label field of conjunction
               or assertion.  Not relevant here */
          S8:  If X is not a TARGET variable, then go to
               S10.
          S9:  /* accumulate definition entries */
               Set #DEF = #DEF +1;
               Set DEF(i), DONE(i) = 1; /* true */
               /* Set TVAL2 (#DEF) = # of dimensions of
               X */
         S10:  /* end of looping */
     S11: For i = 1 to #DEF, perform steps S12 to S29.
```

S12: Get ith definition entry (via TVAL (i));
/* If TVAL2(i) > 0, then X an array */

S13: If current STMT.TYPE is not a diagnosis then go to S16.

S14: Set N = # of LOGIC entries referencing the diagnosis;
Set TEST_IDS (j) = the test label of jth LOGIC entry, for j = 1 to N;
Go to S17;

S15: Set N = 1;
Set TEST_IDS(1) = the test label of the waveform;

S17: For k = 1 to N, perform steps S18 to S26.

S18: Set TEST_ID = TEST_TDS(k).

S19: For j = 1 to #SE, performs steps S20 to S26.

S20: If current (via jth storage entry) statement type is not diagnosis, then go to S23.

S21: If DEF(j) = 1 and X is not in the other parameters of the diagnosis, then go to S26.

S22: Search all LOGIC entries referencing the diagnosis: If there exists a LOGIC entry in the test module with label = TEST_ID, then go to S25; else go to S26.

```
            S23:  If DEF(j) = 1, then go to S26; if X is
                  not in the SOURCE variable list, then
                  add X to the list.
            S24:  If the waveform is not in the module with
                  label = TEST_ID, then go to S26.
            S25:  /* accumulate local references */
                  Set #REF = #REF +1;
                  Set REFERENCE (#REF) = j;
                  Set DONE(j) = 1.
            S26:  /* End of looping j */
      S27:  /* Accumulate definition and save its local
            references */
            Set ND = ND +1;
            Set DEFN (ND) = TVAL(i);
            Set REFL (ND) = #REF;
      S28:  /* end of looping k from S17 */
S29:  /* end of looping i from S11 */
S30:  /* determine scope of X; global or local */
      Set SCOPESW = 0; /* 0 = local; 1 = global */
      Set K = 0.
S31:  For i = 1 to #SE, is every DONE(i) = 1?
      If yes, then go to S33.
      S32:  /* some residual global references, so X
            global */
            Set SCOPESW = 1;
            Add each entry with DONE(i) = 0 to the stack
            REFERENCE, and mark its SCOPE field for X as
            global; go to S37.
      S33:  For i = 1 to ND while (SCOPESW = 1), perform
            the step S34.
            S34:  Set j = REFL(i);
```

```
                If k = j, then set SCOPESW = 1;
                else set k = j.
        S35:    If SCOPESW = 0, then go to S43.
        S36:    /* get all reference entries */
        S37:    For i = 1 to ND, perform steps S38 to S41.
                S38:  If i > 1 and DEFN(i) = DEFN(i -1),
                      then go to S41.
                S39:  Mark the SCOPE field for X as global.
                S40:  /* Output XREF and ATR entry */
                S41:  /* end of looping i from S37 */
        S42:    Go to S44
        S43:    /* local variable(s) */
                /* For i = 1 to ND, output XREF & ATTR
                entries */
        S44:    If X has been used in two test modules
                or more, and also in two diagnoses or
                more, then output error message #11.
        S45:    /* end of looping from S0 */
S46:    Return;
```

TABLE 6.4 STEPS IN DIGRAPH CREATION AND ANALYSIS, AND SEQUENCE DETERMINATION

STEP NAME		SUMMARY OF TASKS
1.	Create Weighted Adjacency Matrix W	Determine total number (n) of nodes in digraph; assign node number to each entry; create nxn matrix W (initialized to zeros)
2	Enter Precedence Relationships (by type numbers) into matrix W	Search every precedence relationship between a predecessor and successor and enter its type to W.
3	Perform Graph Analysis	Create (unweighted) adjacency matrix A from W; analyze W and/or A to ensure that no error conditions exist (except possible cycles)
4	Cycle Detection and Elimination	Create path matrix from A; search for possible cycles; delete the cycles if possible; otherwise report as error to user.
5	Sequence Determination	Rank the nodes of the digraph according to precedence, and then reorder the nodes by their rank.

6.3 INTRA-TEST-MODULE ANALYSIS AND SEQUENCING

This section presents in greater detail the subphases of intra-test-module graph creation and analysis, and sequence determination. It also provides the logical errors that may be detected by each subphase; the corresponding messages are referred to by the reference numbers of Table 6.3.

Each subphase must be repeated once for every test module in the whole NOPAL test specification.

6.3.1 CREATE WEIGHTED ADJACENCY MATRIX, W

The set of names for the waveforms (conjunctions or assertions), the diagnoses, and the variables (local or global) appearing in a test module form the rows and columns of the weighted adjacency matrix, W.

Algorithm 6.2 gives the steps of creating the weighted adjacency matrix for a given test module. It begins with the determination of the size of the matrix. Then it assigns node numbers to all the conjunctions and assertions, diagnoses, and the variables in the test module. Next it allocates the matrix. Finally it initializes the matrix to all zeros, indicating no relationship between any pair of nodes as a default. The matrix is now ready for entering precedence relationships.

ALGORITHM 6.2 CREATE WEIGHTED ADJACENCY MATRIX FOR A TEST MODULE

S1: /* Calculate the size of the matrix W */

Let #W, #D, and #V be respectively the number of waveforms (i.e., conjunctions and assertions), the number of diagnoses, and the number of variables (local or global) in the given test module;

Set N = #W + #D + #V.

S2: /* Assign node numbers */

Successively assign node numbers (1 through N) to each entity (waveforms, diagnoses, and variables);

Create back-and-forth linkage pointers so that if a node number is given then the storage entry for the corresponding entity is readily accessible, and vice versa.

S3: Allocate the Weighted Adjacency Matrix W as an N x N matrix.

S4: /* initialize W to 0 */

Set W (i,j) = 0, (for all i, j).

S5: Return.

6.3.2 ENTER PRECEDENCE RELATIONSHIPS INTO MATRIX W

Two types of precedence relationships may exist among the entities in a test module. Data determinacy relationships (type 1) exist between a waveform or diagnosis which defines a variable X as TARGET and the variable X, also between a variable Y and a waveform or diagnosis which uses the variable Y as SOURCE. The principle that a data must be generated before it can be used is thereby guaranteed. Waveform setup relationship (type 2) exists between the stimulus conjunction and the measurement conjunction. This ensures that stimuli are applied before measurements can be made under normal condition.

The data determinacy relationship is mandatory and always enforced, hence it is associated with the highest priority, 1. The waveform setup relationship is implied only if the measurement conjunction does not generate a variable which is in turn used in the stimulus conjunction. Therefore, this relationship is associated with a lower priority 2.

Algorithm 6.3 shows how these two types of relationships are detected and entered in the weighted adjacency matrix, W. If both the stimulus conjunction and measurement conjunction are not null, then type 2 (waveform setup relationship) is entered in the matrix W in the row

ALGORITHM 6.3 DETECT AND ENTER WAVEFORM SETUP AND DATA DETERMINACY RELATIONSHIPS FOR A TEST MODULE

S0: Define data determinacy relationship as precedence type 1, priority 1, and waveform setup relationship as precedence type 2, priority 2.

S1: /* waveform setup relationship */

If stimulus conjunction or measurement conjunction is null, then go to S2.

S1.1: Let i and j be the node numbers assigned to the stimulus conjunction and measurement conjunction respectively.

S1.2: Set W(i,j) = 2.

S2: /* data determinacy relationships in waveforms */

For each waveform w in the test module, perform steps S2.1 to S2.3.2.

S.2.1: Let i be the node number assigned to w.

S.2.2: /* waveform to data */

For each TARGET variable V in the waveform w, perform steps S2.2.1 to S2.2.2.

S2.2.1: Let j be the node number assigned to V.

S2.2.2: Set W(i,j) = 1.

S2.3: /* Data to waveform */

For each SOURCE variable V in the waveform w, perform steps S2.3.1 to S2.3.2.

S2.3.1: Let j be the node number assigned to V.

S2.3.2: Set W(j,i) = 1.

S3: /* data determinacy relationships in diagnoses */

For each diagnosis d used in the test module, perform steps S3.1 to S3.3.2.

S3.1: Let i be the node number assigned to d.

S3.2: /* operator input variables */

For each operator response variable V in the diagnosis d, perform steps S3.2.1 to S3.2.2.

S3.2.1: Let j be the node number assigned to V.

S3.2.2: Set W(i,j) = 1.

S3.3: /* other parameters */

For each variable V in d's other-parameters field, perform steps S3.3.1 to S3.3.2.

S3.3.1 Let j be the node number assigned to V.

S3.3.2 Set W(j,i) = 1.

S4: Return.

and the column corresponding to the stimulus conjunction and the measurement conjunction respectively (steps S1 to S1.2). Then for each variable (corresponding to node i) in a waveform (conjunction or assertion, corresponding to node j), a data determinacy relationship (type 1) is entered in the matrix W, in the row i and the column j if the variable is defined as TARGET, or in the row j and column i if the variable is used as SOURCE (steps S.2 to S.2.3.2). This indicates that a waveform defining a TARGET variable is a predecessor of the variable, and similarly a waveform using a SOURCE variable is a successor of the variable. Finally, for each variable (corresponding to node i) in a diagnosis (corresponding to node j), a data determinacy relationship (type 1) is entered in W, in the row i and column j if the variable is an operator input variable or in the row j and column i if the variable is one of the other parameters in the diagnosis (steps S3 to S3.3.2). This means that an operator input variable in a diagnosis plays the same role as a TARGET variable in a waveform, and a variable in the other parameters of a diagnosis plays the same role as a SOURCE variable in a waveform.

6.3.3 GRAPH ANALYSIS OF ADJACENCY MATRIX FOR A TEST MODULE

After all the precedence relationships have been

entered into the weighted adjacency matrix, the matrix may optionally be printed out for user's inspection. For example, Figure 6.4 shows the matrix for the test module FREQ in the sample test specification MINIRADIOSET of Figure 3.2. To the left of the matrix are node numbers, entity names, and entity types (conjunction, assertion, diagnosis, or variable).

The weighted adjacency matrix, W, is now ready for further analysis to detect possible logical errors. But to speed up processing and for the purpose of cycle detection in the next stage, an adjacency matrix, A, corresponding to W is generated as follows:

A(i,j) = 1 if W(i,j) > 0;

= 0 otherwise.

Four types of analysis which are performed at this stage are summarized in the following:

(a) If there exist multiple TARGET definitions for a variable, then a warning (Message #3) is sent to the user indicating that they must be under mutually exlusive condition. This is detected by examining the column in A for each variable. If such a column has two or more entries of 1's, i.e., given i be the node number for the variable, there exist j and k such that $j \neq k$ and A(j,i) = A(k,i) = 1, then the warning message is sent.

INTRA MODULE SEQUENCING FREQ
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7	8	9
1	$S_W0001	CONJUNCTION	0	2	0	0	0	0	0	0	0
2	$M_W0001	CONJUNCTION	0	0	0	0	0	0	0	1	1
3	$M_W0002	ASSERTION	0	0	0	0	0	0	0	0	0
4	D4	DIAGNOSES	0	0	0	0	0	0	1	0	0
5	D5	DIAGNOSES	0	0	0	0	0	0	0	0	0
6	D6	DIAGNOSES	0	0	0	0	0	0	0	0	0
7	VAR1	VARIABLE	0	1	1	0	0	0	0	0	0
8	F1	VARIABLE	0	0	1	0	0	1	0	0	0
9	V1	VARIABLE	0	0	0	0	0	0	0	0	0

0 = no relationship

1 = data determinacy relationship

2 = waveform setup relationship

FIGURE 6.4 WEIGHTED ADJACENCY MATRIX FOR TEST MODULE FREQ.

(b) If there are two or more TARGET variables declared in an assertion, then it is an error of ambiguity. The way to detect this error is to examine the row in A for each assertion. If such a row has more than one 1-entry, i.e., given i be the node number for the assertion there exist j and k such that $j \neq k$ and $A(i,j) = A(i,k) = 1$, then the error message is sent to the user (Message #4).

(c) If an assertion has a TARGET variable but the relation operator is not an equal sign ('='), then a warning of inconsistency is sent to the user (Message #7), and the system takes an action of setting the relation operator to a equal sign.

(d) If an assertion has a TARGET variable X, but the expression preceding the equal sign ('=') in the assertion does not match the variable X, then an error of ambiguity is issued (Message #6).

Note that cases (c) and (d) can be detected by first examining the row for an assertion in the adjacency matrix A to determine the number of TARGET variables defined in the assertion. If there exists no TARGET variable in the assertion, then both cases (c) and (d) need not be checked. If there is more than a TARGET

variable, then case (b) must have already detected and (c) and (d) should be skipped. Thus, if there is exactly only one TARGET variable in an assertion, then the assertion statement must further be retrieved to check the relational operator and the expression preceding the operator.

If any errors have been detected during this stage, the Processor proceeds to the next subphase of cycle detection, but will skip the last subphase of sequence determination.

6.3.4 CYCLE DETECTION AND ELIMINATION

This subphase deals with another important type of analysis of the digraph. It performs the tasks of cycle detection, enumeration, and elimination. This is needed in order to 1) detect cycle and eliminate them automatically if possible, and 2) report to the user about erroneous circular definitions.

In order to do such analysis, a path matrix (or reachability matrix) of the digraph is generated. A path matrix, P, is a nxn matrix consisting of 1's and 0's, with a 1 in row i and column j if and only if there is a "path" from the node i to node j, i.e., node j can be "reached" from node i by tracing the edges of the digraph.

Formally, a path in a digraph is a sequence of edges of the form $(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)$. The path is said from node v_1 to v_k and of length (k-1). A path is simple if all edges and all nodes on the path, except possibly the first and the last nodes, are distinct. A cycle is a simple path of length at least 1 which begins and ends at the same node [AHO 74]. Finally, a path matrix, P, for a digraph can be defined by the adjacency matrix as follows:

$P(i,j) = 1$ if $A(i,k_1)=A(k_1,k_2) = \ldots = A(k_m, j)$
$= 1$, for some $k_1, k_2, \ldots, k_m$, and m:
$= 0$ otherwise.

Equivalently, P can be defined as

$$P = A + A^2 + \ldots + A^n$$

where n is the total number of nodes in the digraph and '+' denotes logical disjunction (OR operation).

A more efficient way of generating the path matrix from an adjacency matrix is Warshall's algorithm, which is as follows:

ALGORITHM 6.4 WARSHALL'S: CREATE PATH MATRIX

S1: Set P(i,j) = A(i,j), for all i,j.
S2: Set j=1
S3: Set i=1
S4: If P(i,j) = 0 then goto S6.

S5: For k=1 to n, set P(i,k) + P(j,k).

S6: Set i = i+1;

If i < = n then goto S4.

S7: Set j = j+1;

If j < = n then goto S3.

S8: Return.

In the above algorithm, n is the total number of nodes in the digraph (i.e., the size of the matrix A or P), and '+' in the step S5 denotes logical OR.

Once the path matrix, P has been created, whether there are cycles in the digraph can be easily detected by examining the diagonal of P. If there is at least one 1-entry on the diagonal, then it means there exists some cycles in the digraph; otherwise the digraph is cycle-free (called acylic digraph). If there are no cycles, then the Processor proceeds to the next sub-phase of sequence determination. Otherwise, each distinct cycle must be identified and examined. If all the edges in the cycle are of precedence priority 1 (in the case of internal analysis it turns out to be precedence type 1 only), then it is an error of circular definition and the corresponding message including those entities in the cycle is sent to the user (Message #5). If there exists an edge of lower priority (i.e., priority number greater than 1), then the edge is removed from the digraph. Eventually either the digraph becomes cycle

free or all the unbreakable cycles are detected and reported to the user. The Algorithm 6.5 identifies and enumerates distinct cycles, and tries to break them if possible. It is expanded from the Algorithm CYCLES in [RIN 76] which was adapted from [BER 71, FLO 67]. The expansion involves the automatic cycle elimination once a cycle is identified (steps S24 to S24.7 of Algorithm 6.5).

The algorithm has five inputs: the total number of nodes in the digraph, n; the adjacency matrix, A; the path matrix, P; the weighted adjacency matrix W; and the precedence priority vector PRIORITY. The first three parameters are needed in enumerating all distinct cycles; while the last two parameters are required, in addition, in the process of cycle elimination. The algorithm determines all cycles by the basic principle that node i is in a cycle with node k if A(i,k) & P(k,i) = 1, i.e., there is an edge from node i to node k and a path from k back to i. From each node, it successively grows a tree by adding a tree edge (i,k) such that A(i,k) & P(k,i) = 1. Whenever a terminal node (i.e., a leaf) of a tree coincides with the root of the tree, the path from the root to the leaf forms a distinct cycle.

After a cycle is found, the algorithm tries to eliminate it by deleting an edge which has lowest

ALGORITHM 6.5 CYCLE ENUMERATION AND ELIMINATION

S1: Set ROOT = 1.

S2: /* initializations: S2 to S6 */

Set REACHJ(k) = ROOT, for each k = ROOT to n.

S3: Set USED(k) = 0, for each k = ROOT to n.

S4: Set LEVEL = 1.

S5: Set PATH(1) = ROOT.

S6: Set i = ROOT.

S7: /* Test if path can be extended with nodes in a cycle:

S7-S11 */

If REACHJ(i) > n then go to step S12.

S8: Set j = REACHJ(i).

S9: If A(i,j) & P(j,ROOT) = 1 and USED(j) = 0

then go to step S18.

S10: Set j = j+1.

S11: If j <= n then go to step S9.

S12: /* Backtrack in tree, reset REACHJ + USED: S12-S17 */

Set REACHJ(i) = ROOT.

S13: Set USED(i) = 0.

S14: Set LEVEL = LEVEL - 1.

S15: If LEVEL = 0 then go to step S26.

S16: Set i = PATH(LEVEL).

S17: Go to step S7.

S18: /* extend path: S18-S23 */

Set USED(j) = 1

S19: Set REACHJ(i) = j+1.

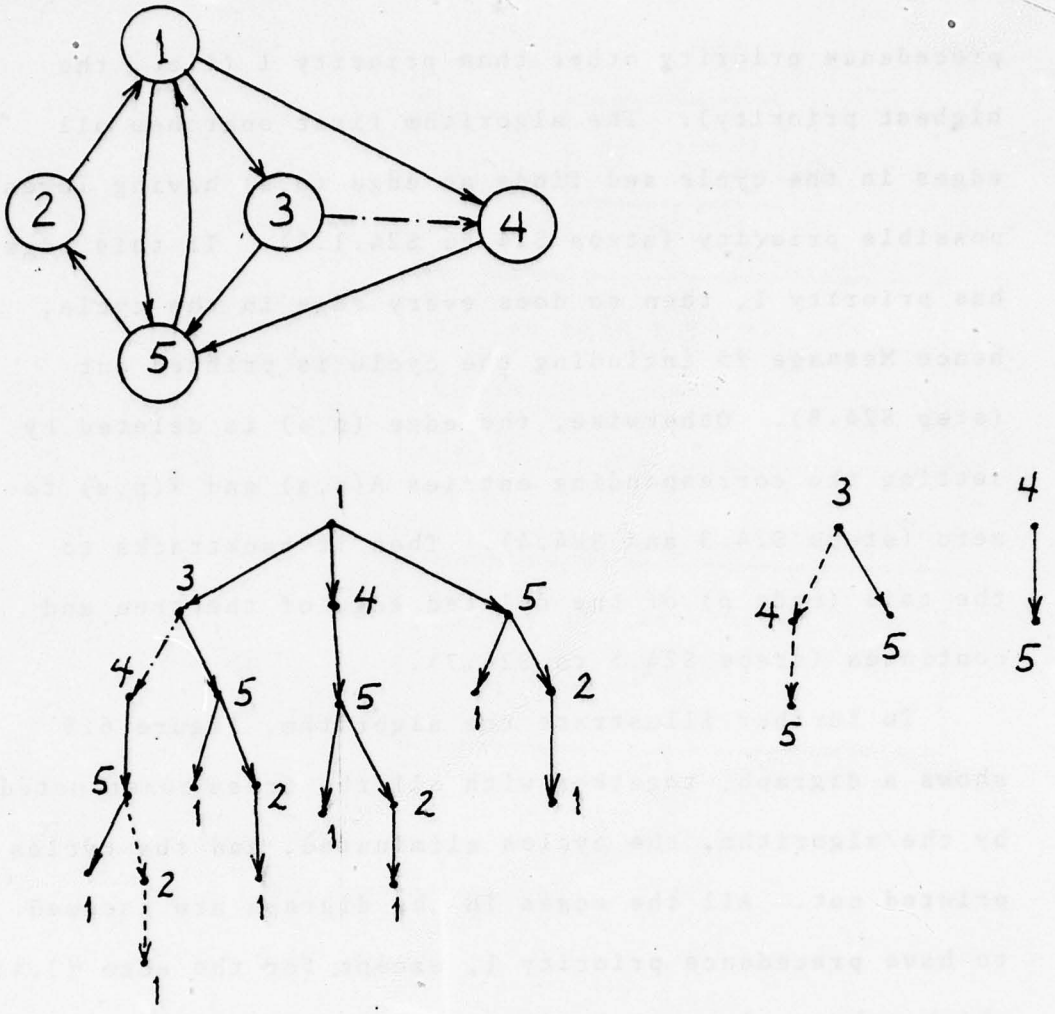
```
S20:    Set LEVEL = LEVEL + 1.
S21:    Set PATH(LEVEL) = j.
S22:    Set i=j.
S23:    If j ≠ ROOT then go to step S7.
S24:    /*Delete an edge of the cycle if possible;
        otherwise print the cycle with an error message:
        S24-S24.8.  Notations used:  pl = lowest pre-
        cedence priority; (p,s) = edge deleted; d =
        level of the node p in PATH; prty = precedence
        priority */
        Set pl = 1.
S24.1:  /* Search each edge in the cycle */
        For k=1 to (LEVEL-1), perform steps S24.1 to
        S24.6.
        S24.1.1:  /* get an edge (p,s) */
                  Set p=PATH(k) and s=PATH (k+1).
        S24.1.2:  /* priority of the edge */
                  Set prty = PRIORITY (W(p,s)).
        S24.1.3:  If prty < = pl then go to S24.1.5.
        S24.1.4:  /* lowest priority; save level */
                  Set pl= prty and d = k.
        S24.1.5:  /* end of looping k */
```

```
S24.2:  If p1 = 1 then  go to S24.8.
S24.3:  /* Get the edge having priority > 1 */
        Set p = PATH(d) and s = PATH(d+1).
S24.4:  /* Delete the edge by zeroing A(p,s) & W(p,s) */
        Set A(p,s) = 0 and W(p,s) = 0.
S24.5:  /* Backtrack the tree to node p (i.e., level d) */
        For k = (d+1) to LEVEL, perform step S24.5.1.
        S24.5.1:  /* Reset USED */
                  Set USED(PATH(k)) = 0.
S24.6:  Set LEVEL = d.
S24.7:  Goto step S16.
S24.8:  /* All edges have priority 1; print the cycle */
        print the cycle:  PATH(k), for k=1 to level
        (Message #5).
S25:    Goto S13.
S26:    Set ROOT = ROOT +1.
S27:    If ROOT < = n then  go to S2.
S28:    Return.
```

precedence priority other than priority 1 (i.e., the highest priority). The algorithm first searches all edges in the cycle and finds an edge (p,s) having lowest possible priority (steps S24 to S24.1.5). If this edge has priority 1, then so does every edge in the cycle, hence Message #5 including the cycle is printed out (step S24.8). Otherwise, the edge (p,s) is deleted by setting the corresponding entries A(p,s) and W(p,s) to zero (steps S24.3 and S24.4). Then it backtracks to the tail (node p) of the deleted edge of the tree and continues (steps S24.5 to S24.7).

To further illustrate the algorithm, Figure 6.5 shows a digraph, together with all the trees constructed by the algorithm, the cycles eliminated, and the cycles printed out. All the edges in the digraph are assumed to have precedence priority 1, except for the edge (3,4) which takes a lower priority, say, 2. The dotted tree edges denote the additional ones which would have been constructed by the algorithm if the edge (3,4) had not been deleted. Note that the algorithm works on the cycles and prints them in ascending order of the node numbers. This example is adapted from [BER 71].

Having created the adjacency matrix, analyzed it for consistency and completeness, and enumerated/ eliminated cycles, the Processor finishes the phase of



CYCLES ELIMINATED: 13451, 134521

CYCLES PRINTED: 1351. 13521, 1451, 14521, 151, 1521

Figure 6.5 Cycle Enumeration and Elimination of a Sample Graph

analysis. If there are no logical errors, such as inconsistencies, ambiguities, incompleteness, and illegal cycles, detected in this phase, the Processor proceeds to the subsequent phases of sequence determination and code generation. Otherwise, the Processor stops processing and asks the user to correct the errors in the test specification and resubmit. In this case, it has provided the user with various reports pinpointing the causes of the problems and suggestions for corrections.

6.3.5 INTRA-TEST-MODULE SEQUENCE DETERMINATION

The task in this subphase is to analyze the cycle-free digraph, represented by weighted and unweighted adjacency matrices, for the purpose of determining the sequences of events in a test module according to precedence relationships and then to reorder the nodes based on their rank.

One way of sequencing the nodes in an acyclic digraph is summarized in Figure 6.6. It assigns to the current rank all the nodes which have no incoming edges from other nodes. Then, it deletes all such nodes (including the corresponding edges) from the digraph. If the resulting digraph is empty, then the algorithm stops with all nodes already assigned to some rank value. Otherwise, it

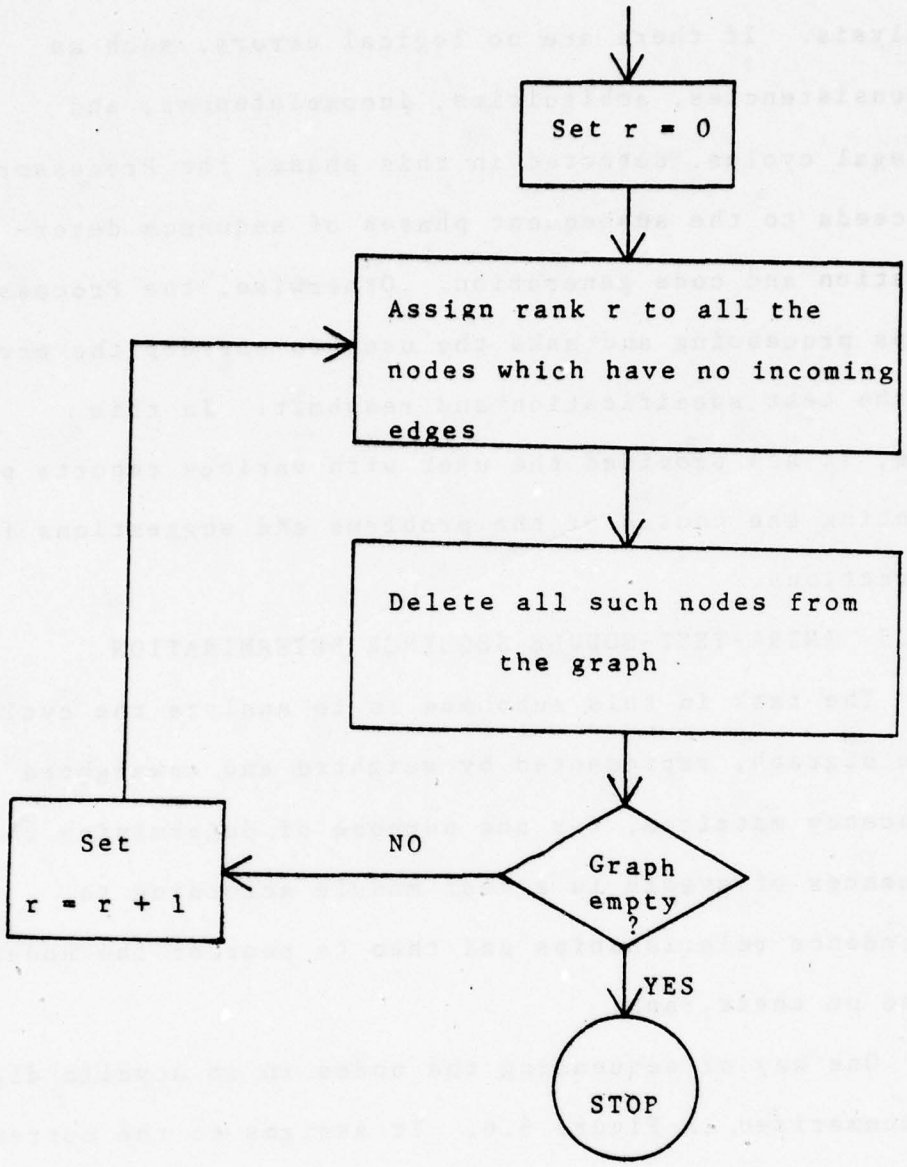


Figure 6.6

Sequential Rank Assignment to the Nodes of Acyclic Digraph

increments the current rank and repeats the process. This algorithm can easily be implemented non-pictorially by using an adjacency matrix and a "flag" vector, as shown in Algorithm 6.6.

Another method of ranking the nodes of a digraph is to multiply the adjacency matrix, A, by itself successively [e.g., BER 71]. At each stage, the nodes which have all-zero column are assigned to the current rank. Then, the current rank is incremented and the process is repeated.

A variation from Algorithm 6.6 is used in the current implementation, and is presented in Algorithm 6.7. This is adopted from the algorithm PRECED [RIN 76], which the current author happened to help develop and implement. The algorithm starts by finding all the nodes with no incoming edges, and assigning them to rank 0(step S2). Then, the nodes of rank (i+1) are all the nodes which are the direct descendants of some nodes of rank i (steps S4 to S7). Note that the rank of some nodes may dynamically be updated. After all the nodes have been partitioned into "rank sets", the algorithm then proceeds to re-arrange the nodes according to their rank (steps S12-S18). The final result is an "order vector" ORDER, where ORDER(i) is the node number which is to be executed at step i.

ALGORITHM 6.6 NON-PICTORIAL IMPLEMENTATION OF FIGURE 6.6

S1: /* Initialize "flag" vector, USED, and level */

Set USED(i) = 0, for all i = 1 to n;

Set level = 0.

S2: /* Initialize old set R1 to contain all nodes with no incoming edges */

Let R1 = $\{ j \mid A(i,j) = 0, \text{ for all } i = 1 \text{ to } n \}$

S3: /* All nodes in R1 are ranked as "level", and flagged */

Set RANK(j) = level and

Set USED(j) = 1, for all j=1 to n.

S4: /* Increment rank level */

Set level = level + 1;

If level = n then go to S9.

S5: /* Create new set R2, consisting of all non-flagged direct descendants of R1 */

Let R2 = $\{ j \mid A(i,j) = 1 \ \& \ USED(j) = 0, \text{ for some } i \text{ in } R1 \}$

S6: If R2 is empty, then go to S10.

S7: /* Update old set R1: all nodes in R2 which have no incoming edges from unflagged nodes */

Let R1 = $\{ j \in R2 \mid A(i,j) = 0 \text{ or } USED(i) = 1, \text{ for all } i \}$

S8: Go to S3.

S9: Return. /* Error: there exists a cycle */

S10: Re-arrange the nodes according to their rank.

S11: Return.

ALGORITHM 6.7 PRECEDENCE SEQUENCE DETERMINATION

S1: /* Initialize rank vector RANK to zeros */

Set RANK(i) = 0, for i=1 to n.

S2: /* Old set R1 initially consists of all nodes having no predecessors */

Let R1 = $\{ j \mid A(i,j) = 0, \text{ for all } i = 1 \text{ to } n \}$

S3: If R1 is empty then go to S11.

S4: Set level = 1.

S5: /* New set R2 consists of all nodes which are the direct successors of some nodes in R1 */

Let R2 = $\{ j \mid A(i,j) = 1, \text{ for some } i \text{ in } R1 \}$

S6: /* All nodes in the new set R2 are ranked "level" */

S7: Set RANK(j) = level, for each j in R2.

S8: If R2 is empty then go to S12.

S9: /* Make the new set the old set */

Set R1 = R2.

S10: Set level = level + 1.

If level $<$ n then go to S5.

S11: /* error: some cycles exist in digraph */

Return.

S12: /* Normal exit: rearrange nodes according to RANK */

Set k = 0.

S13: For i = 0 to (level-1), perform steps S14 to S18.

S14: For j = 1 to n, perform steps S15 to S18.

S15: If RANK(j) $\neq$ i then go to S18.

S16: Set k = k+1. 264

S17: Set ORDER(k) = j.

S18: /* end of looping i and j */

S14: Return.

Figure 6.7 shows the rank sets and order vector produced by applying this algorithm to the corresponding adjacency matrix of Figure 6.4 (which in turn corresponds to a test module in the sample problem of Figure 3.2). Since nodes of the same rank level can come in any order, i.e., they can occur in parallel, the sequence of the nodes generated here from the algorithm is not unique.

This sequence of nodes is the backbone subsequently used in the next phase of code generation for each test module.

SEQUENCE OF PROCESSING FOR TEST FREQ

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	1	0	$S_W0001	CONJUNCTION	(<J24_B, GND> = CONST_S(27.5 VOLT));
2	4	0	D4	DIAGNOSES	TYPE = #5, TIME= 0.00000E+00, RESPONSE=(VAR1);
3	5	0	D5	DIAGNOSES	AFFECTED COMPONENTS = FREQ_TOL(STD_5MHZ_FRE), OTHER PARAMETERS=('FREQ'), TYPE = #6;
4	7	1	VAR1	VARIABLE	LOCAL
5	2	2	$M_W0001	CONJUNCTION	(<J22, GND> = SINE_D(V1 VOLT ,F1 HZ ,VAR1 SEC)) TARGET: F1, V1 SOURCE: VAR1;
6	8	3	F1	VARIABLE	LOCAL
7	9	3	V1	VARIABLE	GLOBAL / TARGET /
8	3	4	$M_W0002	ASSERTION	IF VAR1=60 THEN F1 = 5E+06 +- 60 ELSE F1 = 5E+06 +- 2.5 SOURCE: VAR1, F1;
9	6	4	D6	DIAGNOSES	OTHER PARAMETERS=(' HZ', F1), TYPE = D;

FIGURE 6.7

SEQUENCED NODES OF DIGRAPH FOR TEST MODULE FREQ.

6.4 INTER-TEST-MODULE ANALYSIS AND SEQUENCING

This section deals with the creation and analysis of the digraph for the whole NOPAL test specification, and the determination of execution sequence of test modules. All the logical errors that may be detected in this phase are also properly identified.

In this process of inter-test-module analysis and sequencing, each test module is considered as an integral unit. Although, various information in each test module may be collected and analyzed to determine precedence relationships. Various precedence relationships with their recognition rules have been summarized in Table 6.1. This set of rules is by no means exhaustive, but extensible to include additional strategies that may be found useful. Each precedence relationship corresponds to a row in the table. It is identified by a precedence type, and associated with a priority and a strategy name. Then its existence between a predecessor and a successor is specified, possibly with an execution time condition. A relationship with an execution time condition means that this condition must be tested dynamically during execution time. These relationships are then used to form a digraph. Each node of the graph represents a global data (variable), a diagnosis, or a test module. Each

directed edge denotes one of these precedence relationships. Based on this graph, the consistency, completeness, and ambiguity of the specification can be checked. Also, test modules thereby are ordered in proper execution sequence.

The six sequencing strategies in Table 6.1 are briefly explained in the following.

1) Data determinacy incorporates the principle that data must be generated before it can be used. The generation of data by a predecessor test module is recognized by the declaration of a TARGET variable. A successor test module references the same variable, declared as SOURCE. This relationship is designated as type 1, and is mandatory. Therefore, it is associated with a highest priority 1.

2) Interactiveness relationships are dictated by the need to exchange messages interactively with the ATE operator. Its predecessor is a diagnosis, and successor a test module, which is connected with the diagnosis by a logical operator "After" (A) or "After-not" (A¬). Thus, two precedence types 2 and 3 are given, which correspond to the use of logical operator A and A¬ respectively. This relationship is mandatory and conditional on actual selection (or non-selection) of the diagnosis at run time. Once the predecessor diagnosis is selected (or not selected), the successor

test module should be executed next.

3) Component protection is based on the concept that non-destructive testing can be achieved if a critical component is tested before other components which depend on it for their normal operation. Hence, the failure of such a critical component will prohibit further testing for those dependent components. This relationship is derived from the information on the protection field in the UUT Component Failures specification. This is mandatory and run-time conditional.

4) Fault isolation strategy schedules tests in a top-down fashion using component subset relationships. The more generic fault isolation tests are performed first. The lower level, more specific tests are then executed or skipped, depending upon whether the failure is detected at the top level. In this relationship, the predecessor is a diagnosis, D, and the successor is a test module whose set of affected components is a subset of the set of affected components diagnosed by D. There are two precedence types (5 and 6) in this class. In type 5, the diagnosis D specifies a set of affected components in disjunction. For instance, if D is selected, the component a or b or c is in failure. In this case, a test module that may result in diagnosis of a subset of components, say, a and/or b, is executed next to isolate the faults more specifically. On the

other hand, if the diagnosis D is not selected, then such a test module will not be scheduled for execution. In the other type 6, the diagnosis D specifies a set of affected components in conjunction (e.g., a and b and c). If D is selected, then it means that components a and b and c are defective. Therefore any test module which may result in diagnosing a subset of components, say, a and/or b, will not be executed since the faults have already been isolated. Otherwise, D is not selected and then such a test will be performed next to further identify the failure. Like precedence types 2, 3 and 4, these two types of fault isolation are conditional with a checking of the predecessor diagnosis at run time.

5) Stimuli application is concerned with efficient application of waveform stimuli. It is based on the assumption that application of stimuli is most time-consuming, hence it is advisable to conduct all the possible tests, once a stimulus is applied. Test modules which have stimuli triplets with higher frequency are predecessors to modules with lower frequency of stimuli triplets.

6) Failure likelihood uses the idea that efficiency is obtained by first testing those components which are more likely to fail. Information is extracted from the failure index field in the UUT Component Failures specification. Type 10, with priority 4 is

given to this relationship.

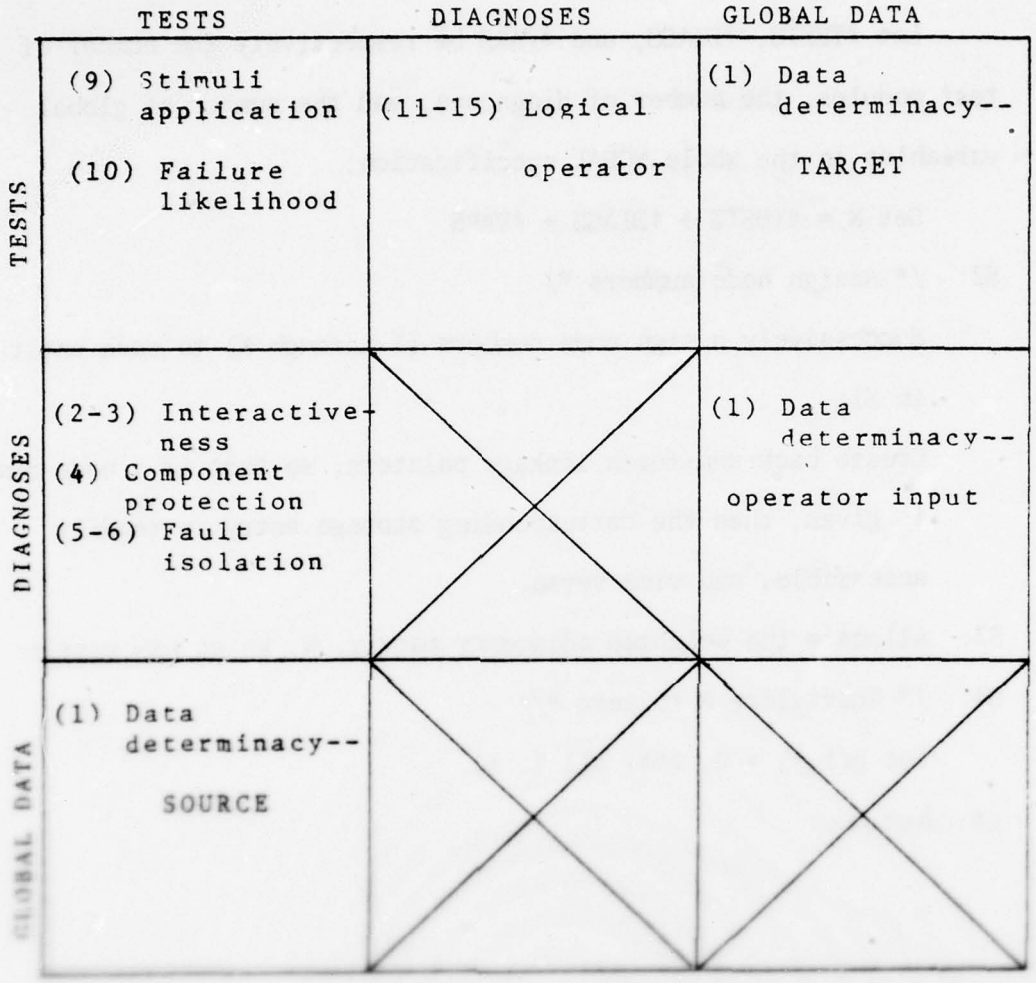
In addition to the types of relationships discussed in the above six strategies, five other types, 11 through 15, are given between a test module and a diagnosis which is selected in the test module by logical operators * (don't-care), |(or), |¬ (or-not), &(and), and &¬ (and-not) respectively. The basic idea is that a diagnosis is posted after its predecessor test module finishes execution. These five types could be combined into one single type, but they are made distinct in order to speed up later processing.

All these precedence types are extracted from the NOPAL specification and put into a weighted adjacency matrix as discussed in the subsequent subsections.

6.4.1 CREATE WEIGHTED ADJACENCY MATRIX, W

The rows (and columns) of the weighted adjacency matrix, W (corresponding to the digraph of the whole NOPAL test specification) consist of the names for the test modules, the diagnoses, and the global variables. The layout of this matrix with all possible precedence relationships is illustrated in Figure 6.8.

The steps of creating the weighted adjacency matrix are summarized in Algorithm 6.8. This is the same as Algorithm 6.2 except that the matrix represents different set of entities. It first obtains the size, N, of the


TESTS
DIAGNOSES
GLOBAL DATA
(9) Stimuli application
(10) Failure likelihood
(11-15) Logical operator
(1) Data determinacy-- TARGET
(2-3) Interactiveness
(4) Component protection
(5-6) Fault isolation
(1) Data determinacy-- operator input
(1) Data determinacy-- SOURCE

ALGORITHM 6.8: CREATE WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION

S1: /* Calculate size of the matrix W */

Let #TESTS, #DIAGS, and #VARS be respectively the number of test modules, the number of diagnoses, and the number of global variables in the whole NOPAL specification;

Set N = #TESTS + #DIAGS + #VARS

S2: /* Assign node numbers */

Successively assign node numbers (1 through N) to each entity in S1;

Create back-and-forth linkage pointers, so that if a node number is given, then the corresponding storage entry is readily accessible, and vice versa.

S3: Allocate the weighted adjacency matrix, W, as an NxN matrix.

S4: /* Initialize W to zero */

Set W(i,j) = 0, (for all i, j).

S5: Return.

matrix as the total number of the test modules, diagnoses, and global variables in the NOPAL specification (step S1). Then it assigns node numbers to all such numbers to all such entities, allocates the matrix W, and finally initializes W to all zeros (steps S2 to S4).

6.4.2 ENTER PRECEDENCE RELATIONSHIPS INTO MATRIX, W

All types of precedence relationships as listed in Table 6.1 are extracted from the NOPAL test specification and entered into the weighted adjacency matrix, W, in the following subsections (6.4.2.1 through 6.4.2.6).

6.4.2.1 ENTER DATA DETERMINACY RELATIONSHIPS

Data determinacy relationships (type 1, priority 1) are entered into the weighted adjacency matrix W between a test module defining a global variable and the variable, also between a global variable and a test module which references the variable.

Algorithm 6.9 shows how data determinacy relationships are detected and entered in the matrix W. Each test module t (corresponding to node i) in the whole NOPAL test specification is examined for the presence of global variables (steps S1 to S7.2). For each TARGET variable (corresponding to node j) in a waveform (i.e., conjunction or assertion) of t, type 1 is entered into

ALGORITHM 6.9 DETECT AND ENTER DATA DETERMINANCY RELATIONSHIPS

S0: Designate data determinacy relationship as type 1, priority 1.

S1: For each test module t in the NOPAL specification, perform steps S2 to S7.2.

S2: Let i be the node number assigned to the test module t.

S3: For each waveform w in test module t, perform steps S4 to S5.2.

S4: /* TARGET global variables */

For each TARGET variable v in the waveform w, perform steps S4.1 to S4.2.

S4.1: Let j be the node number for the variable v.

S4.2: If SCOPE of v is global, then set W(i,j) = 1.

S5: /* SOURCE global variables */

For each SOURCE variable v in the waveform w, perform steps S5.1 to S5.2.

S5.1: Let j be the node number for the variable v.

S5.2: If SCOPE of v is global, then set W(j,i) = 1.

S6: For each diagnosis d in test module t, perform steps S7 to S7.2.

S7: /* SOURCE global variables used in diagnosis */

For each variable v in the other-parameters of the diagnosis d, perform steps S7.1 to S7.2.

S7.1: Let j be the node number for v.

S7.2: If SCOPE of v is global, then set W(j,i) = 1.

S8: For each diagnosis d in the NOPAL specification, perform steps S9 to S10.2.

S9: Let i be the node number for the diagnosis d.

S10: /* Operator input variables -- TARGET */

For each variable v in the operator response variable list of the diagnosis d, perform steps S10.1 to S10.2.

S10.1: Let j be the node number for the variable v.

S10.2: If SCOPE of v is global, then set W(i,j) = 1.

S11: Return.

matrix W in row i and column j to indicate that the test module t is a predecessor and the variable a successor in the relationship of data determinacy (steps S4 to S4.2). Also, for each SOURCE variable (corresponding to node j) in a waveform or diagnosis of the test module t, type 1 is entered into W in row j and column i, indicating that the variable is a predecessor of t (steps S5 to S7.2). Finally, for each diagnosis corresponding to node i) in the NOPAL specification, if its operator input variable (corresponding to node j) is global, then type 1 is entered into W in row i and column j to denote that the diagnosis is the predecessor.

6.4.2.2 ENTER INTERACTIVENESS AND LOGICAL OPERATOR RELATIONSHIPS

Interactiveness relationships are entered between a diagnosis and a test module connected by logical operator A (type 2) or A¬ (type 3). The remaining logical operators (*, |, |¬, &, &¬) are used to enter logical operator relationships (types 11 through 15) between test modules and diagnoses connected by the respective logical operators.

Algorithm 6.10 shows the procedure to detect and enter these relationships. In each test module (corresponding to node i), the logic-diagnosis list is examined (steps S1 to S5). Depending on the logical operator

ALGORITHM 6.10 DETECT AND ENTER INTERACTIVENESS AND LOGICAL OPERATOR RELATIONSHIPS

S0: Designate interactiveness relationships as type 2 for logical operator A, and type 3 for A¬. Also designate logical operator relationships as types 11 through 15 for logical operators *, |, |¬, &, and &¬, respectively.

S1: For each test module t in the NOPAL specification perform steps S2 to S5.

S2: Let i be the node number for test module t.

S3: For each operator-diagnosis pair (op, d) in the logic-diagnosis list of t, perform steps S4-S5.

S4: Let j be the node number for diagnosis d.

S5: If op = '?' then set W(j,i) = 2; /* A */
else if op = '?¬' then set W(j,i) = 3;
/* A¬ */
else if W(i,j) ≠ 0 then give error (Message #10);
else if op='*' then set W(i,j) = 11;
else if op = '|' then set W(i,j) = 12;
else if op = '|¬' then set W(i,j) = 13;
else if op = '&' then set W(i,j) = 14;
else if op = '&¬' then set W(i,j) = 15;

S6: Return.

in each logic-diagnosis pair, different types of relationships are entered between the test module and the diagnosis (corresponding to node j). If the operator is A (after) or A¬ (after-not), then type 2 or 3 is entered into matrix W in row j and column i to indicate that the diagnosis is a predecessor of the test module by interactiveness relationship. Otherwise, type 11, 12, 13, 14, or 15 is entered into W in row i and column j according as the operator is *, |, |¬ , &, or &¬ respectively. In this case, the diagnosis is considered to be a successor of the test module.

6.4.2.3 ENTER COMPONENT PROTECTION RELATIONSHIPS

Algorithm 6.11 shows how the component protection relationships between diagnoses and test modules are extracted and entered into the weighted adjacency matrix W. For each affected component, a set T of test modules each of which includes it as an affected component is obtained first (steps S1 to S3). For each element in the component protection list of this same affected component, another set D of diagnoses each of which includes the element as an affected component is computed next (steps S4 to S5). Finally, for each diagnosis (corresponding to node i) in D, and for each test module (corresponding to node j) in T, type 4 is entered into the matrix W in row i and column j indicating that

ALGORITHM 6.11 DETECT AND ENTER COMPONENT PROTECTION RELATIONSHIPS

S0: Designate component protection relationships as type 4, priority 1.

S1: For each affected component c in the UUT specification of component failures, perform steps S2 to S11.

S2: If component-protection list of c is null then go to S11.

S3: Let Tc be the set of test modules which include c as one of the affected components.

S4: For each affected component p in c's component-protection list, perform steps S5 to S10.

S5: Let Dp be the set of the diagnoses which include p as an affected component.

S6: For each diagnosis d in Dp, perform steps S7 to S10.

S7: Let i be the node number for diagnosis d.

S8: For each test module t in TC, perform steps S9 and S10.

S9: Let j be the node number for t.

S10: If W(i,j) = 0, then

Set W(i,j) = 4;

else give warning (Message #9).

S11: /* end of looping each c */

S12: Return.

the diagnosis precedes the test module by component protection relationship.

6.4.2.4 ENTER FAULT ISOLATION RELATIONSHIPS

Fault isolation (i.e., component subset) relationships are detected and entered into the matrix W between a diagnosis and a test module whose set of affected components is a subset of that of the diagnosis. If the diagnosis identifies affected components in disjunction, then type 5 is entered in the proper entry of W. Otherwise, it diagnoses components in conjunction and a type 6 is entered.

Algorithm 6.12 gives the steps of detecting and entering these relationships. First, for each test module t in the whole NOPAL specification, a set C_t of distinct affected components in all the diagnoses of the test module t is computed (steps S1 and S2). Then, for each diagnosis (corresponding node i) in the NOPAL specification, another set D of affected components in this diagnosis is obtained (steps S3 and S4). If these components are in conjunction, then precedence type is set to 6; otherwise 5 (step S7). Finally for each test module t (corresponding to node j), if C_t is not empty and C_t is a proper subset of D, then the precedence type is entered into the matrix W in

ALGORITHM 6.12 DETECT AND ENTER FAULT ISOLATION RELATIONSHIPS

S0: Designate fault isolation relationships as types 5 and 6, both with priority 2.

S1: /* Compute affected-components set for each test module */

For each test module t in the NOPAL specification, perform step S2.

S2: Let C_t be the set of distinct affected components in all diagnoses of the test module t.

S3: /* Compute affected-components set for each diagnosis, and enter component subset relationships */

For each diagnosis d in the NOPAL specification, perform steps S4 to S11.

S4: Let Cd be the set of affected components in the diagnosis d.

S5: If Cd is null, then goto S11.

S6: Let i be the node number for the diagnosis d.

S7: If d's affected components are conjunctive then set ptype = 6;

else set ptype = 5.

S8: For each test module t in the NOPAL specification, perform steps S9 and S10.

S9: Let j be the node number for t.

S10: If C_t is not null then if C_t is properly contained in C_d then set W(i,j) = ptype.

S11: /* end of looping each diagnosis */

S12: Return.

row i and column j to indicate that the diagnosis is a predecessor of the test module.

6.4.2.5 ENTER STIMULI APPLICATION RELATIONSHIPS

Stimuli application relationships (type 9) are entered into the matrix W between test modules such that the predecessor test module has more frequently used stimuli than the successor does.

Algorithm 6.13 shows how these relationships are extracted and entered in the matrix W. It begins with the calculation of stimuli triplet frequencies (steps S1 to S2.4). The frequency of a stimuli triplet is defined as the total number of occurrences of the triplet in all the test modules. Then the algorithm indexes all stimuli triplets in decreasing order of their frequencies (steps S3 to S3.2). Next it obtains the index value of the most frequent stimuli triplet (i.e., the triplet having lowest index value) in each test module (steps S4 to S4.8). Finally, for each pair of test modules (corresponding to nodes i and j) if the index value of the most frequent stimuli triplet in test module i is smaller than that in test module j, then a type 9 is entered into the matrix W in row i and column j to indicate that test module i is a predecessor by stimuli application relationship.

ALGORITHM 6.13 DETECT AND ENTER STIMULI APPLICATION RELATIONSHIPS

S0: Designate stimuli application as type 9, priority 3.

S1: /* Calculate stimuli triplet frequencies:

Let s be a stimuli triplet (i.e., <conn_dim_ex> = <func_dim_ex>). then the frequency of s:

Freq(s) = total number of occurrences of s in all test modules */

For each test module t in the NOPAL specification, perform steps S2 to S2.4.

S2: For each stimuli triplet s in test module t, perform steps S2.1 to S2.4.

S2.1: If s is not yet in the stimuli triplets stack, then add s to it and set the Freq field to 0.

S2.2: Let p be the position of s in the stack.

S2.3: /* Increment frequency */

set Freq(p) = Freq(p) + 1.

S2.4: Add to the triplet list of test module t a cell pointing to p of the stack.

S3: /* Index all stimuli triplets in the order of decreasing frequencies, except that index "null stimuli" 1 */

Set Index(1) = 1. /* null stimuli */

S3.1: Sort stack entries by their frequencies (in decreasing order of Freq).

S3.2: Set "Index" fields of the sorted stack to 2, 3, ... successively.

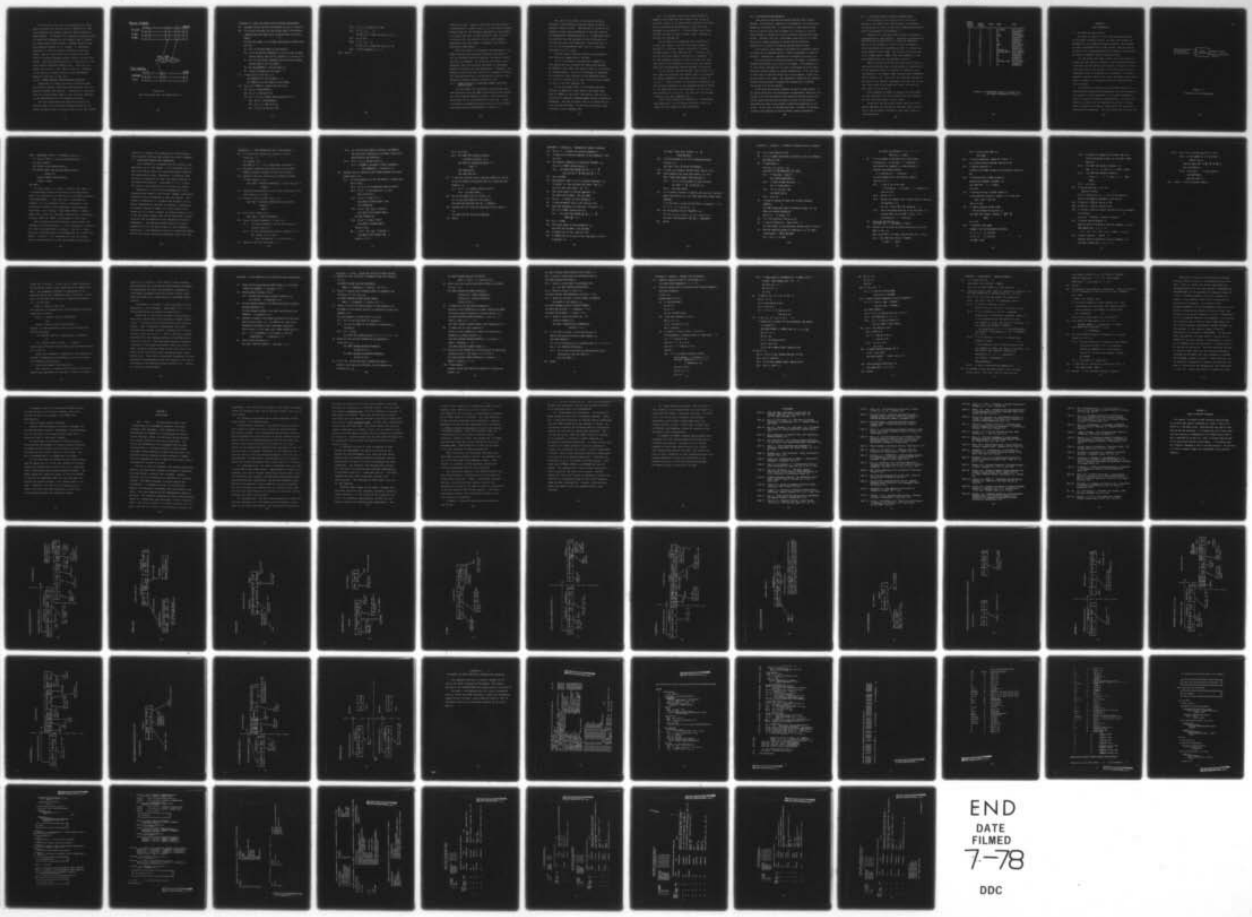
S4: /* get index value of the most frequent stimuli triplet in each test module */

For each test module t, perform steps S4.1 to S4.8.

```
        S4.1:  Let i be the node number for t.

        S4.2:  Let lowindex = Very-large#.

        S4.3:  Set P = Listhead (i).

        S4.4:  While p ≠ null, perform steps S4.5 to S4.6
                S4.5:  If low-index > Index (Left)
                              set lowindex = Index (Left)
                S4.6:  Set p = right.

        S4.7:  If lowindex = Very-large# then
               set lowindex = 0.

        S4.8:  Set lowix (i) = lowindex.

S5:     /* Enter type 9:  For all i,j,
        If Index (most frequent stimuli triplet of test i)
        < Index (most frequent stimuli triplet of test j)
        then W(i,j) = 9          */
        For i=1 to (#TESTS - 1), perform steps S5.1 to S5.4.
        S5.1:  Set Ix = Lowix (i).
        S5.2:  For j = (i+1) to #TESTS, perform steps
               S5.3 to S5.4.
               S5.3: Set Jx = Lowix (j).
               S5.4:  If Ix < Jx then
                             set A(i,j) = 9
                      else if Ix > Jx then
                             Set W(j,i) = 9.

S6:     Return.
```

AD-A054 910 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 14/2
AUTOMATIC TEST PROGRAM GENERATION.(U)
MAR 78 Y K CHANG DAAA25-75-C-0650
UNCLASSIFIED 77-02 ECOM-75-0650-F-1 NL

4 OF 4
AD
A054910



END
DATE
FILMED
7-78
DDC

To illustrate the steps in Algorithm 6.13, three data structures are used as shown in Figure 6.9. One is a stack for all stimuli triplets, which consists of three fields: 1) "Triplet" for storing a stimuli triplet, 2) "Freq" for frequency of the triplet, and 3) "Index" for index value of the triplet. Another stack for all test modules consists of two fields: 1) "Listhead" pointing to a list of stimuli triplets in each test module, and 2) "Lowix" which will contain the index value of the most frequent stimuli triplet in each test module. The last data structure is a collection of list cells used to construct linked lists. There are two fields in each cell. The "Left" points to a location in the stack for stimuli triplets, and the "Right" points to next cell in the list.

6.4.2.6 ENTER FAILURE LIKELIHOOD RELATIONSHIPS

Algorithm 6.14 shows the steps of detecting and entering the failure likelihood relationships into the matrix W between test modules. Information is extracted from the failure-index field of each affected component in the specification of UUT component failures.

For each test module the algorithm obtains the failure index of the affected component which has the lowest assigned value of failure index in the test module

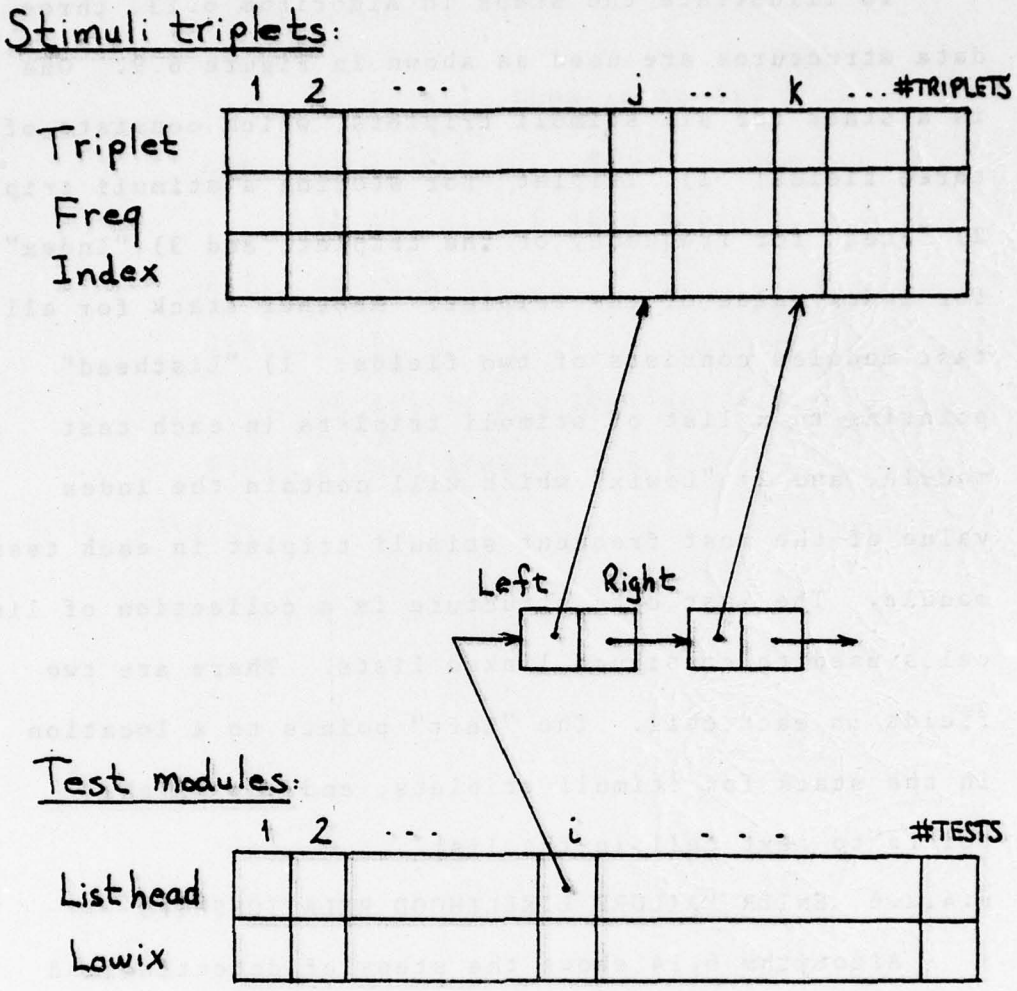


Figure 6.9

Data Structures Used in Algorithm 6.13

ALGORITHM 6.14 DETECT AND ENTER FAILURE LIKELIHOOD RELATIONSHIPS

S0: Designate failure likelihood relationship as type 10, priority 4.

S1: /* For each test module, get the failure index of the affected component which has lowest value of failure-index in the test module */

For each test module t in the NOPAL specification, perform steps S2 to S6.

S2: Let i be the node number for test module t.

S3: If all the affected components in t have not been assigned failure indices, then set FAIL_INDICES(i) = 0 and go to S6.

S4: Let c be the affected component which has lowest assigned failure index in the test module t.

S5: Set FAIL_INDICES(i) = failure index of c.

S6: /* end of looping each test module */

S7: /* Enter type 10 into W(i,j), if

FAIL_INDICES(i) < FAIL_INDICES(j) */

Let #TESTS be the total number of test modules.

S8: For i=1 to (#TESTS-1), perform steps S9 to S19.

S9: Set I_x = FAIL_INDICES(i).

S10: If I_x = 0, then go to S19.

S11: For j = (i+1) to #TESTS, perform steps S12 to S19.

S12: Set J_x = FAIL_INDICES(j).

S13: If J_x = 0, then go to S19.

S14: If $I_x < J_x$ then go to S18.

```
        S15:  If I_x = J_x then go to S19
        S16:  /* I_x > J_x      */
              If A(j,i) = 0 then set A(j,i) = 10.
        S17:  Go to S19.
        S18:  /* I_x < J_x    */
              If A(i,j) = 0 then set A(i,j) = 10.
        S19:  /* end looping j & i  */
S20:  Return.
```

(steps S1 to S6). Then for each pair of test modules (corresponding to nodes i and j), if the lowest assigned failure index of the affected components in test module i is lower than that in test module j, then a type 10 is entered into the matrix W in row i and column j indicating that test module i precedes j by failure likelihood relationship.

After all types of precedence relationships in the NOPAL specification have been extracted and entered into the weighted adjacency matrix, the phase of graph creation is complete. If there are no logical errors detected, then the Processor continues to subsequent phases of analysis and code generation. Otherwise, the Processor stops, and the user has to resubmit his NOPAL test specification after correcting all errors as pinpointed by various reports produced so far.

6.4.3 GRAPH ANALYSIS OF ADJACENCY MATRIX FOR NOPAL SPECIFICATION

At this stage, all precedence relationships have been extracted from the NOPAL specification and entered into the weighted adjacency matrix. This matrix may be printed out at the discretion of the user. Figure 6.2 shows the weighted adjacency matrix for the sample test specification MINIRADIOSET of Figure 3.2.

The task of this phase is to perform further analysis on the weighted adjacency matrix to detect possible logical errors. The problem of cycles in the digraph is dealt with separately in the next section. An (unweighted) adjacency matrix, A, is generated from the matrix W for the purposes of more efficient processing and later cycle analysis. Each entry A(i,j) is 1 if the corresponding entry W(i,j) is non-zero. Otherwise A(i,j) is zero.

Various types of analysis which are performed in this phase are summarized as follows:

(a) If a variable is referenced as SOURCE in a test module but the variable has never been defined as TARGET in any other test module, then it is an error of incompleteness. This can be detected by examining the column for each variable in the adjacency matrix, A. If such a column is all zero, then an error message is sent to the user (Message #1).

(b) On the other hand, if a variable has been defined as TARGET in a test module, but never been used as SOURCE in any other test module, then it is a situation of possible incompleteness, hence a warning is warranted. The way to detect this is to check the row in A for each variable. If the row is all zero, then a warning is sent (Message #2).

(c) If a global variable has been defined as TARGET in more than one test module, then it may be an error, unless it is under mutual exclusive situation. This is detected by examining the column for each variable in the matrix A. If there exist two or more non-zero entries in such a column, then a warning is printed out (Message #3).

(d) If there is no diagnosis ever specified in a given test module, then the user might forget to include it by mistake, hence a warning of possible incompleteness is sent (Message #8). To detect, the row for each test module in matrix A is examined. If all the diagnosis entries of such a row are all zero, then there is no diagnosis at all in the test module, hence the corresponding warning message is produced.

(e) To achieve interactiveness, there should be no more than one successor test module which is connected to a given diagnosis by the same logical operator A (or A$\neg$). This situation is detected by examining the row for each diagnosis in the weighted adjacency matrix W. If there are two or more entries of 2, or two or more entries of 3 in such a row, then an error message is sent to the user (Message #15).

After these checks for consistency, completeness, and unambiguity, the Processor proceeds to next sub-phase of cycle detection and elimination.

6.4.4 CYCLE DETECTION AND ELIMINATION

This section is concerned with another important task of digraph analysis: cycle detection, enumeration, and elimination. All cycles in the digraph are first identified. In a given cycle, if there exists an edge which has a priority other than the highest (i.e., priority 1), then the system proceeds to eliminate the cycle by deleting an edge which has lowest priority. Otherwise, the cycle is unbreakable; hence the user is provided with an error message indicating the illegal circular definitions. Algorithm 6.5 of Section 6.3.4 precisely performs these functions.

In the course of cycle analysis, a path matrix (or reachability matrix) is created from the adjacency matrix A. This is accomplished by using the Warshall's algorithm (i.e., Algorithm 6.4) as presented in Section 6.3.4.

After the path matrix P has been generated, its diagonal is examined to determine the existence of any cycles in the digraph. If the diagonal has all zero entries, then there are no cycles in the digraph and the Processor proceeds to the next subphase of sequence determination. On the other hand, if there exists at least one non-zero entry in the diagonal, it means there are some cycles in the digraph. Then Algorithm 6.5 identifies and enumerates distinct cycles and tries to eliminate them automatically, if possible (see Section 6.3.4 for more details).

At this stage, the Processor completes the phase of graph analysis. If any logical errors (such as inconsistencies, incompleteness, ambiguities, and illegal cycles) have been detected, then appropriate messages together with various reports would have been provided to the user for the purposes of identifying the causes of the problems, possibly with suggestions for correction. In this case, the Processor stops here. Otherwise, the Processor proceeds to next subphase of inter-test-module sequence determination.

6.4.5 INTER-TEST-MODULE SEQUENCE DETERMINATION

This subphase further analyzes the cycle-free digraph of the NOPAL test specification to determine the sequences of events in the whole NOPAL specification based on precedence relationships.

Section 6.3.5 describes this process of sequence determination in more details. Basically the sequencing algorithm (e.g., Algorithm 6.7) takes the adjacency matrix of the digraph to rank the nodes according to their precedence relationships and then to reorder the nodes based on their rank. The output of this process is an "order vector" ORDER, such that ORDER(i) is the node number which will be executed at step i.

Shown in Figure 6.10 are the order vector and the rank sets resulting from an application of Algorithm 6.7 to the adjacency matrix of Figure 6.2 (which in turn corresponds to the sample NOPAL specification MINIRADIO-SET of Figure 3.2). Note that the algorithm generates only one of the possible sequences of nodes, due to the fact that nodes of the same rank level can occur in any order, or in parallel.

This sequence of nodes is used in the next phase of code generation for the invocation of test modules and for inserting proper control logic.

After both the intra-test-module and inter-test-module analysis and sequencing have been done successfully, the Processor proceeds to next major phase of code generation.

ORDER VECTOR INDEX	ORDER VECTOR	RANK	NAME	TYPE
1	1	0	DC-INPUT	TEST MODULE
2	7	1	D2	DIAGNOSIS
3	8	1	D3	DIAGNOSIS
4	5	2	FREQ	TEST MODULE
5	9	3	D4	DIAGNOSIS
6	10	3	D5	DIAGNOSIS
7	11	3	D6	DIAGNOSIS
8	20	3	V1	VARIABLE
9	2	4	AMPL	TEST MODULE
10	12	5	D7	DIAGNOSIS
11	13	5	D8	DIAGNOSIS
12	3	5	DISTORT_2W	TEST MODULE
13	4	5	DISTORT_VOLT	TEST MODULE
14	19	6	30	DIAGNOSIS
15	14	6	24	DIAGNOSIS
16	15	6	25	DIAGNOSIS
17	16	6	26	DIAGNOSIS
18	6	7	DISTORT_10MW	TEST MODULE
19	17	8	27	DIAGNOSIS
20	18	8	28	DIAGNOSIS

FIGURE 6.10 SEQUENCED NODES OF DIGRAPH FOR THE SAMPLE PROBLEM OF FIGURE 3.2

CHAPTER 7

CODE GENERATION

7.1 Introduction and Overview

As shown in Figure 7.1 this code generation phase of the NOPAL Processor accepts as input the digraph represented by the weighted adjacency matrix, the sequence of nodes represented by an order vector, and the storage entries in the simulated associative memory, and produces as output a complete object program in OPAL.

To some extent, the order vector which was generated during the last phase of analysis and sequencing provides a skeleton for the object test program because it identifies each node of the digraph in the order that the corresponding event is supposed to be executed. In order to generate complete object code, more information about each node is needed to be extracted from the simulated associative memory.

As indicated in Figure 6.4 of System Flowchart for Specification Analysis and Sequencing, and Code Generation, the process of code generation is initiated immediately after the phase of sequence determination. For each test module, an intra-test code generation process is begun once to generate a procedure right after the intra-test analysis and sequencing. Once the inter-test analysis and

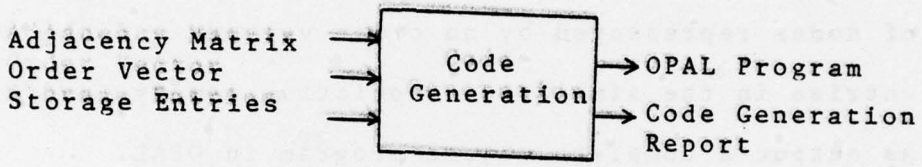


Figure 7.1

Overview of Code Generation

sequencing is done, the inter-test code generation process is started to generate a test monitor (main procedure) which includes defining global and systems variables, defining procedures for messages, invoking test-module procedures, and inserting control logic. The steps of code generation are summarized in Table 7.1. Capitalized names enclosed in parentheses are procedures to be invoked. Note that these steps are general, hence they should be still applicable even if the object test program were not in OPAL.

In order to facilitate the illustration of the code generation process in the subsequent sections, a procedure (subroutine) EMIT with one argument of character string is defined as to output a string of text to an output medium. For example, in terms of PL/1 language, it can be implemented as follows:

```
EMIT:  PROCEDURE (TEXT); /* STREAM-ORIENTED */
   /*  OUTPUT "TEXT" to output medium "OUTFILE" */
   DCL TEXT CHAR(*);
   DCL OUTFILE FILE STREAM OUTPUT;
      PUT FILE (OUTFILE) EDIT (TEXT) (A);
END EMIT;
```

or

TABLE 7.1 STEPS OF CODE GENERATION PHASE OF NOPAL PROCESSOR

A. For each test module, define a procedure (GENTEST).

B. Generate a test monitor, i.e., main procedure:

   1. Define procedure header.

   2. Declare and initialize global variables and systems variables.

   3. For each message defined in the NOPAL specification, define a procedure (GENMSG).

   4. Based on the order vector for the NOPAL specification, generate a sequence of test-module procedure calls and properly insert control logic (INVOKE_TEST).

   5. End the procedure.

C. Merge the user-supplied ATE-function definitions coded in object language, if any.

```
EMIT:  PROCEDURE (TEXT); /* RECORD_oriented  */
   /*  Output "TEXT" to output medium "OUTFILE"   */
   DCL TEXT CHAR(*);
   DCL OUTFILE FILE RECORD OUTPUT;
   DCL OUTREC CHAR (MAX_OUTFILE_RECORD_SIZE)
            BASED (P);
       LOCATE OUTREC FILE(OUTFILE);
       OUTREC = TEXT;
END EMIT;
```

In other words, if a piece of object code, TEXT, is to be generated during the process of code generation, then it is achieved by an invocation of the EMIT procedure, "CALL EMIT(TEXT);". Also note that a concatenation operator (||) is frequently used to concatenate a sub-string with another in the subsequent code generation algorithms. For instance, "CALL EMIT (TEXT1 || TEXT2);" means emit a piece of code which consists of two character strings, TEXT1 and TEXT2, concatenated in that order.

The following sections describe in more detail the subphases of code generation. In the subsequent discussions the object language OPAL is assumed. Section 7.2 deals with the intra-test-module code generation for a given test module. Section 7.3 is concerned with the inter-test-module code generation for the whole NOPAL specification.

Section 7.4 presents the capability of incorporating user-supplied ATE functions written in object language.

7.2 INTRA-TEST-MODULE CODE GENERATION

This subphase is concerned with the internal code generation for a test module. It is initiated once for each test module, after the intra-test-module analysis and sequencing is done. Basically, a procedure (subroutine) is generated for each test module defined in the NOPAL specification. Algorithms 7.1 to 7.3 give the steps of code generation for a given test module.

Algorithm 7.1 starts with determination of the dimensions and bounds of all global, TARGET, subscripted variables which are defined in this test module. The number of dimensions of a subscripted variable can be easily determined by checking its number of subscripts. In current version of NOPAL, the lower bound of any dimension in an array variable is always assumed to be 1. As for the upper bounds, they can be determined by examining all the occurrences of the respective subscript expressions. The maximum possible value in a given dimension will be the upper bound for that dimension. Algorithm 7.1 then generates code for a subroutine header and for any local variable declarations (Steps S2 & S3).

ALGORITHM 7.1 CODE GENERATION FOR A TEST MODULE, t

```
S0:  /* Initialize counter for iterative control
     functions  */
     Set #LOOPS = 0;
     Set ENDS = '$'.  /* $ = OPAL stmnt end marker */
S1:  Determine dimension and bounds for each global
     TARGET variable defined in this test module t.
S2:  Generate code for subroutine (procedure) header
     of test module t, e.g.
        CALL EMIT ('DEFINE SUBROUTINE '|| test-label(t) ||
                        ENDS);
S3:  Generate declarations for all local variables in the
     test modules t, if any.
S4:  Declare a local system flag, e.g.
        CALL EMIT ('DECLARE SYSFLAG BOOLEAN INITIAL
                     (TRUE)'||
                       ENDS);
S5:  Start test timing, e.g.
        CALL EMIT ('START SYSTIME$');
S6:  Issue 0-timing diagnoses, if any, e.g.
     S6.1:  For each diagnosis d in the best module t,
            perform step S6.2.
     S6.2:  If the corresponding logical operator = '*'
            and timing (d) = 0 then
                CALL POSTDIAG(d); /* issue diag. */
S7:  Generate code for waveforms, e.g.
```

S7.1: Let n be the total number of entities, and ORDER be the order vector determined in the phase of intra-test-module analysis and sequencing.

S7.2: For i = 1 to n, perform step S7.3.

S7.3: If ORDER(i) represents a node of waveform w,

CALL GENWAVE(w); /* code for w */

S8: Generate code for issuing non-zero-timing diagnoses with proper control logic, e.g.

S8.1: For each diagnosis d in the best module t, perform steps S8.2 to S8.9.

S8.2: Let op be the corresponding logical operator.

S8.3: If op = '*' and timing (d) > 0, then

CALL POSTDIAG(d).

S8.4: If op = '|' then /* OR */

Call EMIT('IF SYSFLAG THEN'); and

CALL POSTDIAG(d).

S8.5: If op = '|¬' then /* OR-NOT */

CALL EMIT('IF NOT SYSFLAG THEN');

and CALL POSTDIAG (d).

S8.6: If op = '&' then /* AND */

CALL EMIT('IF SYSFLAG THEN');

and go to S8.9.

S8.7: If op = '&¬' then /* AND-NOT */

CALL EMIT('IF NOT SYSFLAG THEN ');

and go to S8.9.

S8.8: go to S8.1.

S8.9: Call EMIT('SET NO_TESTS_IN_CONJ(d)

= NO_TESTS_IN_CONJ(d)-1 $');

Call EMIT('IF NO_TESTS_IN_CONJ(d) = 0

THEN ');

CALL POSTDIAG(d);

CALL EMIT('END IF$').

S9: If there are iterative control functions (STEP_OF or LIST_OF) in the test module, then generate code for closing the loops properly, e.g.

s9.1: For i = 1 to #LOOPS, perform step S9.2.

S9.2: CALL EMIT('END$');

S10: Set the test-module flag to "tested", e.g.

Let i be the node number for test module t;

CALL EMIT('SET TEST_FLAG(' || i || ') = TESTED$');

S11: Generate code for closing the subroutine for test module t, e.g.

CALL EMIT('END' || test_label(t) || ENDS);

S12: Return.

ALGORITHM 7.2 POSTDIAG(d): PROCEDURE FOR ISSUING A DIAGNOSIS

S1: Set k = 0. /* counter for affected components */

S2: If there are no affected components in the diagnosis d, then go to S6.

S3: Save affected components in system area "SYSCOMP", e.g.

For i=1 to #COMPS, perform step S3.1.

S3.1: Call EMIT('SET SYSCOMP(' || i || ') = ''' ||
FAIL_FN(i) || '(' || COMP_ID(i) || ')' $');

S4: Set k = #COMPS.

S5: Save operator ("and" or "or") in affected components, e.g.

If AND_OR = '&' then CALL EMIT('SET SYSOP = "AND" $');

else call EMIT ('SET SYSOP = "OR "$').

S6: Save actual number of affected components, e.g.

Call EMIT('SET NO_COMPS = ' || k || ENDS);

S7: Set k=0. /* counter for other parameters */

S8: If other-parameters field is null then go to S11.

S9: Save other parameters in system area "SYSPARM", e.g.

For i=1 to #PARMS, perform step S9.1.

S9.1: Call EMIT('SET SYSPARM(' || i || ') = ' ||
PARM(i) || ') $');

S10: Set k = #PARMS.

S11: Save actual number of other parameters, e.g.

Call EMIT('SET NO_PARMS = ' || k || ENDS);

S12: If message type is null, then go to S15.

S13: If TIMING field is not null or zero, then delay issuance of message, e.g.

```
Call EMIT ('DELAY UNTIL SYSTIME  >= ' ||
                TIMING || ENDS);
```

S14: Issue the message by call the corresponding message routine; e.g.,

```
Call EMIT ('CALL ' || OP_MSG.TYPE || ENDS);
```

S15: If there is no operator variable input, then go to S17.

S16: Issue command to get input from console; e.g. call

```
EMIT ('INPUT FROM CONSOLE ' || OP_RPS.VAR(1));
```

S16.1: For i=2 to total-number-of-input-variables,

```
Call EMIT (',' || OP_RPS.VAR (i));
```

S16.2: Call EMIT (ENDS);

S17: If Y/N operator response is expected, then wait until operator's manual intervention, e.g. Call EMIT('DELAY UNTIL MANUAL INTERVENTION$);

```
/* In this case, a system variable SYSY_N is assumed to be set
to 'Y' or 'N' afterwards   */
```

S18: Set the diagnosis flag to "SELECTED", e.g.

```
Let i be the node number for the diagnosis d;
Call EMIT ('SET DIAG_FLAG (' || i || ') = SELECTED$');
```

S19: Return.

ALGORITHM 7.3 GENWAVE(w): PROCEDURE TO GENERATE CODE FOR A WAVEFORM

S1: If w is null then go to S24.

S2: If w is a simple conjunction or assertion, then call SIMPLE(w); and then go to S24.

S3: /* conditional */

Generate if_clause, e.g.

call EMIT('IF ' || CONDITION(w) || 'THEN');

S3.1: /* True-part: Simple */

Call SIMPLE(TRUE_PART(w));

S3.2: /* Trace through false-part */

Set w = FALSE_PART(w);

S3.3: If w is not null, then

call EMIT('ELSE');

S3.4: go to S1.

S4: /* Steps S4 through S23 define the internal procedure SIMPLE(w). */

Let TEMP be push-down stack of character strings, e.g. DCL TEMP (255) VARYING CONTROLLED;

Set #V = 0. /* counter */

S5: If w is an assertion, then go to S11.

S6: /* case of conjunction: steps S6-S10.2 */

For each triplet in the conjunction, perform steps S7 to S10.2.

S7: Save UUT connection points in system area, e.g. call EMIT ('SET NO_PINS = ' || #PTS || ENDS);

S7.1: For i = 1 to #PTS,

```
                Call EMIT('Set SYSPINS(' || i || ') =  '||
                          POINT.ID(i) || ENDS);
```

S8: /* If an argument in the function is of the form:

< relational_op > < arith_expr > [< dimension >]

then replace the argument by a system variable X

and add a new pseudo assertion:

X < relational_op > < arith_expr > */

For each argument arg in the function, perform steps S8.1 to 8.4.

S8.1: If arg is not of the form:

< relational_op > < arith_expr > [< dimension >],

then go to S8.

S8.2: Set #V = #V + 1.

S8.3: Replace the argument with a system variable, retaining the dimension; e.g.

Set arg = 'SYS_V' || #V[|| < dimension >].

S8.4: Push a new pseudo assertion to the stack TEMP; e.g.

Allocate TEMP, and set TEMP = 'SYS_V' ||#V ||

< Relational_op > || < arith_expr >.

S9: Invoke the ATE function; e.g.

```
Call EMIT('CALL' || FUNC_DIM_EX || ENDS);
```

S10: Generate code for those new pseudo assertions in the stack TEMP, if any, e.g.

While stack TEMP is not empty, perform steps S10.1 to S10.2.

```
S10.1:  Call EMIT('SET SYSFLAG = SYSFLAG&'
                  || TEMP  ||   ENDS);
```

S10.2: Pop the stack TEMP, e.g.

Free TEMP;

S11: /* Case of assertions: Steps S11 to S22.3 */

If w involves control function, then go to S17.

S12: /* regular assertion */

If there is no TARGET variable in the assertion, then go to S14.

S13: /* assertion with a TARGET variable */

Generate an assignment statement, e.g.

Call EMIT('SET' || w || ENDS);

Go to S23.

S14: /* assertion having no TARGET variable */

If the assertion involves a "range", i.e., of the form:

exp1 = exp2 +- exp3 [%]

then go to S16.

S15: /* Simple assertion without range;

AND with SYSFLAG for decision */

CALL EMIT('SET SYSFLAG = SYSFLAG & ' || w ||

ENDS);

go to S23.

S16: /* assertion with range;

expand it into two equivalent assertions:

e_1 <= e_2 + e_3 [*e_2/100],

e_1 >= e_2 - e_3 [*e_2/100]. */

Set TEMP = exp3.

```
S16.1:  If percent (%) appears in the range, then treat
        exp3 as percentage of exp2, e.g. Set TEMP = TEMP *
        exp2/100.
S16.2:  Call EMIT('SET SYSFLAG = SYSFLAG &' ||
        exp1  || ' < = ' || exp2 || '+' || TEMP || ENDS);
S16.3:  Call EMIT('SET SYSFLAG = SYSFLAG &'
          || exp1 ||  '  > =' ||  exp2 || '-' ||  TEMP ||
           ENDS);
S16.4:  go to S23.
S17:  /* Control functions:  as of now,
         STEP_OF or LIST_OF                    */
      Let "cv = func_id (e1, e2, ..., en)" be the assertion;
      and n be the number of arguments in the function .
S18:  If func_id = 'STEP_OF' then go to S21.
S19:  If func_id = 'LIST_OF' then go to S22.
S20:  /* In error, unless new control functions is to be introduced
      in the future                                          */
      go to S12.  /* default:  treated as regular */
S21:  /* Control function STEP_OF  */
      Generate code for iteration by step and increment, e.g. Call
      EMIT('REPEAT FOR ' || cv || '='||
        e1 || 'TO' || e2 || 'BY' || e3 ||  ENDS);  Go to S23.
S22:  /*Control function :  LIST_OF   */
      Generate code for iteration by a list of elements, e.g.
      call EMIT ('REPEAT FOR '|| cv || '='  );
```

```
S22.1:  For i=1 to n, perform steps S22.2 to S22.3.
        S22.2:  If i-th element ei is of the form:
                    STEP_OF (a,b,c)
                then call EMIT(a || ' TO' || b || 'BY' || c);
                else call EMIT(ei);
        S22.3:  If i=n then      /* last element */
                    call EMIT(ENDS);
                    ELSE call EMIT(',');
S23:    Return.  /* end of procedure SIMPLE */
```

A run-time local system flag (SYSFLAG), which will contain the TRUE/FALSE result of the test module after execution, is initialized (step S4) and will be properly altered later. The algorithm then starts the test timing, and selects all diagnoses which are associated with logical operator * (don't-care) and with zero timing (steps S5 to S6.2). Next it generates code for conjunctions and assertions by calling GENWAVE (Algorithm 7.3), in the order specified by the order vector (which is a product of last sequencing stage) (steps S7 to S7.3). Then the algorithm continues to generate code for selecting non-zero-timing diagnoses with proper run-time control logic (steps S8 to S8.9). The control logic is primarily conveyed by the test-result flag (SYSFLAG) and varies according to different logical operators. Finally it closes the loops for iterative control functions, if any, sets the test flag, and closes the subroutine definition for the test module (steps S9 to S11).

Algorithm 7.2 is a supporting routine (POSTDIAG) for Algorithm 7.1. It is invoked whenever a diagnosis is ready to be posted. The algorithm first saves the affected components (steps S2 to S7) and other parameters (steps S7 to S11) in some system areas, so that these are accessible in the message routine (to be discussed in next section). Next, the message is issued with proper

time delay (steps S13 and S14). Note that each message (identified by "type") will be defined as a sub-routine, hence is invoked by a subroutine call. The algorithm proceeds to generate code for possible operator response and input variables (steps S15 to S18). Finally it sets the diagnosis flag to "selected" (step S18).

Algorithm 7.3 is another supporting routine (GENWAVE) for algorithm 7.1. It is called whenever a waveform (conjunction or assertion) in a test module is due for code generation. The algorithm generates code for a simple conjunction or assertion by calling an internal routine SIMPLE (steps S2 and S3.1). It also takes care of conditional waveforms (steps S3 to S3.4). The internal procedure SIMPLE (steps S4 to S23) generates code for a simple conjunction or assertion. In the case of conjunction, the procedure first saves the UUT connection points (steps S7 and S7.1). If an argument in a function-dimension-expression is of the form: < relational_operator > < arithmetic_expression >, then the argument is replaced by a new variable X, and a new assertion of the form: X < relational_operator > < arithmetic_expression > is to be added (steps S8 to S8.4). Then the stimuli or measurement function is invoked with proper arguments (step S9), and the code for new assertions, if any, is generated

(steps S10 to S10.2). In the case of simple assertion, the procedure checks if the assertion has a TARGET variable. An assignment statement is generated for an assertion which has a TARGET variable (step S13). Otherwise, if the assertion involves a range specification, i.e., of the form:

```
expr1 = expr2 +- expr3 [%];
```

the assertion is expanded into two equivalent assertions (steps S16 to S16.4):

```
expr1 < = expr2 + expr3 [*expr2/100];
        and
expr1 >= expr2 - expr3 [*expr2/100];
```

Finally, if the assertion is one of the following two control functions:

```
X = STEP_OF (initial, upper-bound, increment);
        or
X = LIST_OF (expr1, expr2, ..., exprn);
```

then proper iteration control logics are generated (steps S17 to S22.3).

After this subphase is completed a test module subroutine is generated corresponding to each test module in the NOPAL specification.

7.3 Inter-test-module Code Generation

This section is concerned with overall inter-test-module code generation as outlined in step B of Table 7.1.

Basically, it defines a "test monitor" main procedure which declares and initializes global/systems variables, defines message routines, and generates a sequence of test-module procedure calls with proper control logic inserted based on the order vector of the NOPAL specification.

Algorithms 7.4 through 7.7 give the steps of code generation in this subphase. Algorithms 7.5 to 7.7 are subroutines which are invoked by Algorithm 7.4.

Algorithm 7.4 essentially defines a main program which will perform the tests described in NOPAL specification. It begins with the generation of main program header. Then it generates declarations for global variables and systems variables, possibly with initializations (steps S2 and S2.1). Next it defines a message subroutine for each message defined in the NOPAL specification (step S3). The algorithm proceeds to generate a sequence of test-module procedure calls with control logic properly inserted, based on the order vector produced in the inter-test-module sequencing phase (steps S4 and S4.1). Note that a subroutine (procedure) has been generated corresponding to each test module in the intra-test-module code generation phase (Section 7.2). Finally it closes the main program.

ALGORITHM 7.4 CODE GENERATION OF TEST MONITOR FOR NOPAL SPECIFICATION

S1: Define object program main procedure header, e.g. call EMIT ('BEGIN OPAL PROGRAM ' || spec_name || '$');

S2: Declare global variables, if any.

S2.1: Declares and initialize systems variables; e.g.,

```
Call INITIAL;   /* Algorithm 7.5 */
```

S3: For each message defined in the NOPAL specification, define a message subroutine, e.g.

For each message m defined in the NOPAL specification, call GENMSG(m); /* Algorithm 7.6 */

S4: Generate a sequence of test-module procedure calls and insert proper control logics, e.g. Let ORDER be the order vector generated in the phase of inter-test-module sequencing;

For each entry i in the order vector ORDER, perform step S4.1.

S4.1: If i corresponds to a test module t, then call

```
INVOKE_TEST(t).   /* Algorithm 7.7 */
```

S5: Close the main procedure, e.g.

Call EMIT ('END OPAL PROGRAM' || spec_name || '$').

ALGORITHM 7.5 INITIAL: DECLARE AND INITIALIZE SYSTEMS VARIABLES

S1: Declare an array, DIAG_FLAG, of diagnosis flags, and initialize it, e.g.

```
Call EMIT('DECLARE DIAG_FLAG BITSTRING(2)
    ARRAY ('|| #diagnoses ||') INITIAL (" 00 ") $');
```

S2: Declare an integer array NO_TESTS_IN_CONJ, for diagnoses, and initialize it to all zero, e.g.

```
Call EMIT('DECLARE NO_TESTS_IN_CONJ INTEGER
    ARRAY ('|| #diagnoses ||') INITIAL (o) $ ');
```

S3: Properly initialize NO_TEST_IN_CONJ(d) of each diagnosis d to the number of test modules which are in conjunction to select the diagnosis, e.g.

For each diagnosis d, perform steps 3.1 to S3.3.

S3.1: Let i be the node number for diagnosis d:

S3.2: Let n be the number of test modules in conjunction in selecting d.

S3.3: If n > 0 then

```
Call EMIT('SET NO_TESTS_IN_CONJ('|| i ||' ) =' || n || '$')
```

S4: Define two flag constants SELECTED and NOT_SELECTED for diagnoses, e.g.

```
Call EMIT('DECLARE SELECTED BITSTRING(2)
        INITIAL("10") $');
Call EMIT('DECLARE NOT_SELECTED BITSTRING(2)
        INITIAL("01")$');
```

S5: /* S5 to S6: declare/initialize variables for tests */

Declare a test flag array, TEST_FLAG, for test modules, and initialize it, e.g.

```
Call EMIT('DECLARE TEST_FLAG BITSTRING(2)
          ARRAY('|| #tests ||') INITIAL("00")$');
```

S6: Define three flag constants NOT_TESTED, TESTED, and SKIPPED for tests, e.g.

```
Call EMIT('DECLARE NOT_TESTED BITSTRING(2)
          INITIAL("00"), TESTED BITSTRING(2)
          INITIAL("10"), SKIPPED BITSTRING(2)
          INITIAL("01")$');
```

S7: /* Common area for UUT affected components */

Declare an array SYSCOMP and two variable SYSOP and NO_COMPS for passing affected components to message routines, e.g.

```
Call EMIT('DECLARE SYSCOMP CHARSTRING('|| max#chars ||')
ARRAY('|| max#comps || ') $');
Call EMIT('DECLARE NO_COMPS INTEGER, SYSOP CHARSTRING(3) $');
```

S8: /* Common area for other parameters */

Declare an array SYSPARM and a counter for passing other parameters to message routines, e.g.

```
Call EMIT('DECLARE SYSPARM CHARSTRING ('|| max#chars ||' )
ARRAY('|| max#parms || ')$');
CALL EMIT('DECLARE NO_PARMS INTEGER $');
```

S9: Declare an array SYSPINS and a counter NO_PINS for passing UUT connection points to ATE stimuli or measurement

```
Call EMIT('DECLARE SYSPINS CHARSTRING('|| max#chars || ')
ARRAY('|| max#pins || ') $');
```

S10: /* Miscellaneous */

Declare a system flag SYSFLAG for dynamically invoking test modules, e.g.

```
Call EMIT('DECLARE SYSFLAG BOOLEAN INITIAL (TRUE) $ ');
```

S10.1: Declare a system timer for performing tests, e.g.

```
Call EMIT('DECLARE SYSTIM REAL$");
```

S10.1: Declare a system timer for performing tests,

e.g. `Call EMIT('DECLARE SYSTIME REAL$");`

S10.2: Define a constant of blank, e.g.

```
Call EMIT('DECLARE SYSB CHARSTRING(1) INITIAL (" ") $');
```

S10.3: Define two variables to contain number of diagnoses and number of test modules, e.g.

```
Call EMIT('DECLARE NO_DIAGS, NO_TESTS INTEGER$');
Call EMIT('SET NO_DIAGS = '|| #Diagnosis || '$');
Call EMIT('SET NO_TESTS = '|| #test || '$');
```

S10.4: Declare a system area SYSY_N for Y/N operator response, e.g.

```
CALL EMIT ('DECLARE SYSY_N CHARSTRING(1)
                    INITIAL(" ") $ ');
```

S11: /* OPAL special feature in specifying UUT pins */

Specify UUT connection points in main program, e.g.

```
Call EMIT('SPECIFY');
```

For each UUT connection point p, perform steps S11.1 to S11.2.

S11.1: `Call EMIT(UUT.POINT_ID(p));`

S11.2: If p is the last UUT point in UUT-connection-points specification, then call `EMIT('$');`
else call `EMIT(',');`

S12: Return.

```
ALGORITHM 7.6  GENMSG(m):  GENERATE CODE FOR MESSAGE m
S1:  Generate subroutine header for the message m, e.g.
     Call EMIT('DEFINE SUBROUTINE '|| m || '$');
S2:  Construct message text by inserting actual affected components
     and other parameters,
     if any, e.g.
     Perform steps S3 to S13.
     S3:  /* initialization */
          Set ip = 0;
          Set is = 0;
          Set len = LENGTH (text).
     S4:  /* Get next character of text */
          Set is = is + 1;
          If is > len then go to S13;
          Set c = text(is);
     S5:  /* check if c is one of the 3 characters:
          left parenthesis, exclamation mark, or single quote    */
          If c = '(' then go to S6;
          else if c = '!' go to S5.1;
          else if c = '''' then go to S5.2;
          goto S2;
          S5.1:  /* ! is a special character in OPAL;
                    Insert another !  (!! denotes '!') */
                    Set text = SUBSTR(text, 1, is-1) || '!!'
                              || SUBSTR(text, is);
                    Set len = len +1;
                    Set is = is + 1;
                    go to S2.
```

```
S5.2:  /* Single quote is represented by " in NOPAL; but by !'
       in OPAL.  Hence change first ' to !   */
       Set text (is) = '!';
       Set is = is + 1;
       go to S2.
S6:    /* Check if (C), (P), (Ci), or (Pi)  */
       Set is = is + 1;
       if is > len then go to S13;
       Set c = text (is);
       If c = 'C' or c = 'P' then go to S7;
                             else go to S5.
S7:    /* Yes, (C), (P), (Ci), or (Pi)      */
       /* Output the text before the left parenthesis, and advance
          text pointer */
          Call EMIT('OUTPUT' || SUBSTR (text, ip, is-ip-1));
          Set ip = is + 1;
          Set is = ip;
          If is > len then go to S10;
          Set d = text (is);
          If d is not a numeric digit, then go to S10.
S8: Set jf = d;
    For is = (is+1) to len, perform steps S8.1 to S8.3.
    S8.1:  Set d = text(is).
    S8.2:  If d is not a numeric digit, then go to S9:
    S8.3:  Set jf = jf*10 + d.
```

```
S9:  Set js = jf;
     Set ip = is;
     goto S11.
S10: /* (C) or (P)   */
     If c = 'C' then set jf = NO_COMPS;
                else set jf = NO_PARMS.
S11: /* Output affected components and other parameters */
     If c = 'C' then set temp = 'SYSCOMP (';
                else set temp = 'SYSPARM (';
     Call EMIT('OUTPUT');
     For j = js to jf, perform steps S11.1 to S11.2.
     S11.1:  Call EMIT(temp || j || ')');
     S12.2:  If j = jf then call EMIT('$');
             else call EMIT(', SYSB, SYSOP');
S12: If is > len then go to S13;
     Set c = text (is);
     If c = ')' then go to S12.1;
                else go to S5.
     S12.1:  Set ip = ip+1;
             go to S4.
S13: /* Output remaining message text */
     If ip <= len, then
     Call EMIT('OUTPUT' || SUBSTR (text, ip) ||
                '$');
S14: Close subroutine definition, e.g.
     Call EMIT('END' || m || '$');
S15: Return.
```

ALGORITHM 7.7 INVOKE_TEST(t): INVOKE TEST MODULE t

```
S1:   /* Initializations     */

      Call EMIT(' SET SYSFLAG = TRUES');

      Let i be the node number for test module t.

S2:   Generate control logic based on predecessor diagnoses, e.g..
      For each diagnosis d, perform steps S2.1 to S2.6.

      S2.1:  Let j be the node number for diagnosis d.

      S2.2:  Set ptyp = A(j,i); /* precedence type */

      S2.3:  If ptyp = 0 then go to S2.6

             else if ptyp = 2 or ptyp = 5 then go to S2.5.

      S2.4:  /* d is related to t by A¬ , component

             protection, or conjunctive component subset */

             If the diagnosis has been selected, then

             set SYSFLAG to false, and skip t's successor, e.g.
             Call EMIT('SET SYSFLAG = SYSFLAG

                AND DIAG_FLAG('||j ||' ) = NOT_SELECTED$');
             Call SKIP(t, 'DIAG_FLAG('||j ||') = SELECTED');
             Go to S2.6.
      S2.5:  /* d is related to t by A, or disjunctive

             component subset                          */

             If the diagnosis has not been selected, then

             set SYSFLAG to false, and skip t's successor, e.g.
             CALL EMIT ('SET SYSFLAG = SYSFLAG and

             DIAG_FLAG ('|| j || ') = SELECTED$');

             Call SKIP(t, 'DIAG_FLAG('|| j || ') =

             NOT_SELECTED');

      S2.6:  /* End of looping each diagnosis */

S3:   If SYSFLAG is true and test module t has not been

      tested before, then emit code for invoking the
```

```
     test module routine, e.g. call EMIT('IF SYSFLAG
     THEN IF TEST_FLAG (' || i ||') = NOT_TESTED
     THEN CALL '|| test_label(t) || '$');
S4:  Return.
S5:  /* Internal routine SKIP(t, condition):  steps S5 to S9. */
     /* If "condition" is true and if t is not "skipped"
     yet, then
     1) Mark t as skipped, and
     2) For each variable v solely defined in t, skip
        recursively each successor test module s which
        uses variables v as SOURCE, by calling SKIP
        (s, condition)                    */
     Let k be the node number for test module t.
S6:  Call EMIT('IF TEST_FLAG(' || k ||' ) #=
          SKIPPED THEN IF' || condition || 'THEN');
     CALL EMIT('SET TEST_FLAG(' || k ||
               ') = SKIPPED$');
S7:  For each TARGET variable v defined in t, perform
     steps S7.1 to S7.2.
     S7.1:  If the column for v in matrix A has exactly
            one non-zero entry (i.e. v is uniquely
            defined as TARGET in t), then perform
            step S7.2, for each test module s using v
            as SOURCE.
     S7.2:  Call SKIP (s, condition);
S8:  /* OPAL special constructs; closing two IF's  */
        Call EMIT('END$  END$');
S9:  Return.  /* for the SKIP internal routine */
```

Algorithm 7.5 declares and initializes systems variables which are used in the generated object program. First it defines two arrays (DIAG_FLAG and NO_TEST_IN_CONJ) and two flag constants (SELECTED and NOT_SELECTED) for the purposes of diagnoses (steps S1 to S4). NO_TEST_IN_CONJ(i) initially contains the number of test modules, which are in conjunction in selecting the diagnosis. At execution time, NO_TEST_IN_CONJ(i) will be decremented once such a test module is executed and results in true or false (depending on the logical operator & or &¬). DIAG_FLAG(i) is set to SELECTED whenever the diagnosis i is selected. The algorithm then defines an array (TEST_FLAG) and three flag constants (NOT_TESTED, TESTED, SKIPPED) for the purposes of test modules (steps S5 and S6). TEST_FLAG is initialized to NOT_TESTED. At execution time, TEST_FLAG(i) for test module i will be set to TESTED once the test module i is performed, or will be set to SKIPPED once the test module i is determined to be omitted for execution. Then the algorithm proceeds to declare common areas for the affected components (SYSCOMP and NO_COMPS) and other parameters (SYSPARM and NO_PARMS) (steps S7 and S8). These variables are set whenever a diagnosis is posted, and then they are used in constructing the message text (see algorithms 7.3 and 7.6). Next, Algorithm 7.5 defines two areas

(SYSPINS and NO_PINS) for passing UUT connection points to ATE stimuli or measurement functions (step S8). Thus SYSPINS contains the set of UUT connection points, and NO_PINS gives the number of these points, both of which are set in a test-module procedure and are accessible when defining ATE stimuli or measurement functions. The algorithm then defines several miscellaneous systems variables (steps S10 to S10.4). Finally it generates a "specify" statement to identify all UUT test points (steps S11 to S11.2). This reflects the OPAL special requirement that UUT points must be specified in main program section rather than in any routine.

Algorithm 7.6 generates code for a message defined in NOPAL. First it defines a subroutine header. Then it constructs the message text by inserting affected components and other parameters (steps S2 to S13). Basically, if any affected components (designated by (C) or (C_i)) have been specified in the original message text, then they have already been saved in a system area (SYSCOMP); hence each corresponding affected component is available from this area and is properly inserted in the message text. Similarly other parameters have been saved in another system area (SYSPARM); hence they are available for insertion in the message text. Finally, the algorithm simply ends the definition for the message subroutine.

Algorithm 7.7 conditionally invokes a test-module routine and inserts necessarily control logic. A flag (SYSFLAG) is first initialized to true (step S1). Then for each predecessor diagnosis the precedence type is examined to generate proper control logic (steps S2 to S2.6). If the diagnosis is related to the test module by precedence type 2 (interactiveness - "A") or 5 (fault isolation - disjunctive component), and if the diagnosis has not been selected, then the flag is set to false, and the successors of the test module are properly skipped for execution (step S2.5). On the other hand if the precedence type is 3 (interactive - "A¬ ") or 4 (component protection) or 6 (fault isolation-conjunctive components) and if the diagnosis has been selected, then the flag is set to false and the successors of the test module are properly skipped (step S2.4). Finally, if the flag is true and the test module has not been tested then the test-module routine is invoked (step S3). An internal routine SKIP (steps S5 to S9) is used to skip successors of a test module conditionally. The condition is either a diagnosis which is selected or a diagnosis which is not selected. If the test module has already been skipped, then the routine will not proceed further. Otherwise, it recursively skips each descendent test module whichever uses a global variable that is uniquely defined in the current test module (steps S7 to S8).

In summary, these four algorithms together define the main program in the object language, which will perform the tests as provided in the NOPAL test specification.

7.4 Object Program Library Inclusion

Usually, the definitions (in object language) for all ATE functions (stimuli, measurement, and evaluation) will be provided in the compile phase (some even at execution time) of the object program.

To facilitate the inclusion of object routines (particularly non-standard or special-purpose routines) the NOPAL system provides the user with an option of including his own object library at this stage. One way of implementing it is summarized as follows:

If the user wants to include his own object libary, then he should put in a file, say, OBJLIB, and provide proper data definition for this file (e.g. a proper DD card in IBM system) when invoking NOPAL processor. The NOPAL system will check the status of the file OBJLIB. If the file is undefined (that is, the user does not provide his object library), then nothing needs to be done. Otherwise, the file will be merged with the object code which has been generated in both intra- and inter-test-module code generation phases.

CHAPTER 8

CONCLUSIONS

This report has described a theory and a methodology for programming computer controlled Automatic Test Equipment. The formalism for this methodology is imbedded in a non-procedural specification language, NOPAL, in which test specifications for various classes of UUTs can be described. The theory and methodology for programming ATE has been incorporated in a software production system (NOPAL Processor) which generates reliable and efficient test programs from test specifications expressed in NOPAL. The test programs are used to analyze and isolate faulty components of the UUT in an ATS.

The use of such an automatic test program generation system in an ATS clearly reduces the amount of expertise and time needed to produce test programs. Equally important is its contribution to confidence in automatic testing. The NOPAL language is non-procedural and incremental. It enables the user to describe test modules in descriptive statements that may appear in any order. Also test modules can be independently added or modified. The Processor analyzes the user's specification, checks for completeness and consistency, deduces sequence of events, and finally produces a reliable, complete test program. The user is relieved of the procedural thinking and

programming. Thus the need for expertise, the amount of program coding and debugging time, and the potential human errors can be much reduced.

The following conclusions can be drawn regarding several other aspects of the NOPAL system:

(1) Man-machine interface: The NOPAL Processor provides many reports to enable the user to check his specifications. The reports include: (a) a reformatted specification listing which reorganizes the specification for better readability and assists the user in understanding his test specification, (b) several cross reference reports to provide the user with a picture of the interactions among various components of the test specification, (c) sequence flowcharts of the resultant test program, and (d) error and warning messages to alert the user to inconsistency, incompleteness, and ambiguities in his specification and to help him pinpoint the trouble spots for correction.

(2) Knowledge base: Two types of knowledge base have been incorporated in the NOPAL system: engineering and computer programming. The former includes identifying failure modes, assuring non-destructive testing, distinguishing stimuli/measurement waveforms from pure computations, and sharing diagnoses among multiple test modules to reduce ambiguity in fault isolation. The latter includes the construction of a directed graph model from the test specification, the analysis of the graph, and the encoding into object test program.

(3) Optimization: The test program generation system automatically sequences the test modules and incorporates control logic in the final test program. The program will dynamically

schedule for execution only those tests needed at test-time. The tests are scheduled in a top-down fashion. The top level, more generic functional test, is first scheduled to discover as many faults as possible. Only when the UUT fails in such a functional test, the program proceeds to perform the lower level, more specific fault isolation tests. Several other optimization methods have also been used. One method reduces the time consumed in application of stimuli. Once a set of stimuli is connected to the UUT points, as many tests as possible are scheduled for execution. Another method uses the idea that efficiency can be obtained by first testing the components which are more likely to fail.

To achieve the goal of total automation of test design and programming for ATE as a solution to the problem of high maintenance cost, both the test determination and the program production processes (top and bottom parts of Figure 1.1) must be automated and then integrated. The automatic test determination process for electronic circuits has been achieved independently in the Moore School by C. Tinaztepe [TIN 77]. The NOPAL language has been used to describe complete test specifications. The language has been found to be useful and adequate.

Although some changes in the object language OPAL have been, and still are, underway, the basic constructs remain unaltered. In the course of this research, two minor, yet important, modifications in OPAL were proposed to ensure the language's ATE independence. One

of them is the capability of passing any name, such as resource, noun, modifier, unit, test point, etc. (see Chapter 3) in a routine, or equivalently assigning any of these items to a variable. The code generation process has been based upon this assumption. Another assumption is that an OPAL subroutine that corresponds to a NOPAL stimulus or measurement function may include a REQUIRE statement. This will guarantee that the characteristics of the required resource can be specified. Except for these two assumptions, the NOPAL features have been able to be realized by the object language OPAL. These additions to OPAL are believed to be very important in manual preparation.

Some of the tool and techniques that have been developed are very powerful and will be generally useful to the developer of programming systems. Most notable is a general purpose statement storage and retrieval subsystem that has been developed.

The three major phases of the NOPAL Processor have been fully designed, including various algorithms and needed data structures. Phase I, syntax and statement analysis, and the intra-test-module analysis and sequencing of phase II has been successfully implemented in PL/I(F) using an IBM 370/168 computer. The inter-test-module analysis and sequencing of phase II has been partially implemented.

A number of research directions may be taken in the future;

(1) Complete implementation: The inter-test-module analysis and sequencing, and the code generation phases of the Processor need to be implemented.

(2) Refinements and extensions; An interactive NOPAL Processor should be a worthwhile endeavor. After all, a user would typically go through several iterations with the NOPAL system until his problem is solved. An interactive system would certainly make the interaction between the user and the system more effective. The two logical values, TRUE and FALSE, should be included in the language. Currently a simple assertion is basically a relational expression. It would be much more flexible if a simple assertion could be extended to include any expression that yields a logical value TRUE or FALSE. More control functions should be investigated to facilitate test modules composition. Variables are considered to be of type REAL in current system. Other types such INTEGER, STRING, and COMPLEX may also be needed. The combination of basic stimuli functions to form a composite one needs to be further studied. Whether the logic part should be extended to include the selection of a diagnosis based on the other diagnoses needs further investigation. Finally it may be worth making the NOPAL syntax more English-like or user-oriented.

(3) Other application domain: The top part of ATS, automatic test determination, has been based on a particular UUT class -- electronic circuits. In principle, the bottom part, automatic test program generation, can be used in describing tests for various classes of UUTs. Other types of UUTs, such as hydraulic and mechanic, should be studied for the purposes of test determination.

In summary the NOPAL language and Processor have automated the test program production process for an automatic test system. Although a lot of further research in this field must be done in the future, this research has made an important step towards the goal of total automation of test determination and test program generation, hence has contributed to the ultimate reduction of the skyrocketing maintenance and support costs of electronics systems.

BIBLIOGRAPHY

[ARI 76] ARINC and IEEE, "IEEE/ARINC standard ATLAS Test Language," IEEE Std. 416-1976, ARINC Spec. 416, Vol. II, IEEE, September, 1976.

[AHO 72] Aho, A.V. and Ullman, J.D., "The Theory of Parsing Translation, and Compiling," Vol. I: Parsing, Prentice Hall, 1972.

[AHO 74] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., "The Design and Analysis of Computer Algorithms," Addison Wesley, 1974.

[BEL 70] Bell Laboratories, "A Decade of ESS," Bell Laboratories RECORD, December 1970.

[BEL 73] Bell Laboratories., "No.4 ESS-Long Distance Switching for the Future," Bell Laboratories RECORD, September 1973.

[BAK 72] Baker, F., "Chief Programmer Team Management of Production Programming," IBM System Journal, 2-1, 1972, pp. 57-73.

[BER 71] Berztiss, A.T., "Data Structures: Theory and Practice," Academic Press, 1971.

[BOE 74] Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973.

[CAR 69] Carr, J.W. and Weiland, J., "A Non-Recursive Method of Syntax Specification," Comm. ACM, Vol. 9, No. 4, 1969.

[CHE 76] Che, H.D. and Chang, Y.K., "The NOPAL Language Specification and User Manual," Moore School Report No. 76-04, University of Pennsylvania, August 1976.

[COD 62] CODASYL Development Committee, "An Information Algebra Phase I Report," Comm. of the ACM, Vol. 5, No. 4, April 1962.

[CON 63] Conway, M.E., Design of Separable Transition Diagram Compilers," Comm. of ACM, July 1963.

[COU 73] Cougar, J.D., "Evolution of Business Systems Analysis Techniques," Computer Surveys, Vol. 5, No. 3, Sept. 1973.

[DEO 74] Deo, N., "Graph Theory With Applications to Engineering and Computer Science," Prentice Hall, 1974.

[ELE 74] Eleccion, M., "Automatic Testing: Quality Raiser, Dollar Saver," IEEE Spectrum, August 1974, pp. 38-43.

[FLO 67] Floyd, R.W., "Non-Deterministic Algorithms," Journal of ACM, Vol. 14, No. 4, October 1967.

[FRA 76] Frankford Arsenal, "Operation Performance Analysis Language (OPAL), Definition, Syntax and Semantics of," Proposed MIL-STD-1462, Change 2, September 1976.

[FRA 77] Frankford Arsenal, "Operation Performance Analysis Language (OPAL)," Proposed MIL-STD-1462, Change 3, January 1977.

[FRE 72] French, A., "A Syntax Analysis Program Generator," Master Thesis, Computer and Information Science, University of Pennsylvania, 1972.

[GAN 75] Gana, J.E., "Use and Extensions of an Automatic Program Generator System For Model Building in the Social and Engineering Sciences," October 1975, Working paper, Moore School, University of Pennsylvania.

[GEN 73] General Dynamics, Electronics Division, SCATE Mark IV, 1973.

[GRE 73] Green, A.M. and Rytter, L.J., "Computer Aided Test Generation For Analog Circuits," Wescon, 1973, 10/3.

[GRI 66] Grindley, C.B., "SYSTEMATICS - A Nonprogramming Language For Designing and Specifying Commerical Systems for Computers," Computer Journal, August 1966.

[HAX 73] Hax, A.C. and Martin, W.A., "Automatic Generation of Customized Model Based Information Systems for Operations Management," Data Base, Vol. 5, Nos. 2,3 and 4, 1973.

[IBM 61] IBM, "Study Organization Plan Documentation Techniques," C20-8075, 1961.

[IBM 71] IBM, "The Time Automated Grid System (TAG): Sales and Systems Guide," GY20-0358-1, May 1971.

[IBM 72] IBM System/360 Operating System, PL/I(F), Language Reference Manual, GC28-8201-3, and Programmer's Guide, GC28-6594-8, 1962.

[LAN 63] Langefors, B., "Some Approaches to the Theory of Information Systems," BIT 3, 1963.

[LIG 74] Liguori, F., ed., "Automatic Test Equipment: Hardware, Software, and Management," IEEE Press, 1974.

[LUS 73] Lustig, J. and Goodman, D.M., "Trends in the Development of Automatic Test Equipment," Proj. SETE Report 210/106, NASA, June 1973.

[LYN 69] Lynch, H.J., "ADS: A Technique in Systems Documentation," Database, Vol. 1, No. 1, Spring 1969.

[MAR 74] Martin, W.A., "OWL: A System For Building Expert Problem Solving Systems Involving Verbal Reasoning," Notes From Course 6871, MIT, Fall 1974.

[MAR 76] Martin, W.A. and Bosyj, M., "Requirements Derivation in Automatic Programming," Proceedings of the Symposium on Computer Softman Engineering, April 1976.

[MAL 75] Malhorta, A., "Design Criteria For A Knowledge Based English Language System For Managements: An Experimental Analysis," MIT Thesis, February 1975.

[MCA 71] McAleer, H.T., "A Look At Automatic Testing," IEEE Spectrum, Vol. 8, May 1971, pp. 63-78.

[MIL 71] Mills, H., "Top Down Programming In Large Systems," Debugging Techniques in Large Systems, Randall Rustin, Ed., Prentice Hall, 1971, pp. 41-45.

[NAS 73] NASA, John F. Kennedy Space Center," Ground Operations Aerospace Language (GOAL)," Textbook TR-1228, April 1973.

[NUN 71] Nunamaker, J.F., "A Methodology for the Design and Optimization of Information Processing Systems," AFIPS Proc., 1971, SJCC, pp. 283-294.

[NUN 76] Nunamaker, J.F., et al, "Computer-Aided Analysis and Design of Information Systems," Comm. of ACM, Vol. 19, No. 12, 1976.

[PRY 74] Prywes, N.S., "Automatic Generation of Software Systems -- A Survey," Database, Vol. 6, No. 2, Fall 1974.

[PRY 75] Prywes, N.S., "Automatic Computer Program Generation For Automatic Testing Systems (ATS)," Report FCF-3-75, U.S. Army, Frankford Arsenal, Philadelphia, Pa., January 1975.

[PRY 77a] Prywes, N.S., "MODEL II - Description and User Manual," February 1977, submitted to IRS under contract No. TIR-17-62.

[PRY 77b] Prywes, N.S., "Automatic Generation of Computer Programs," Advances in Computers," Vol. 16, M. Rubinoff and M. Yovits, Eds., Academic Press, 1977, in print.

[RAM 73] Ramirez, J.A., "Automatic Generation of Data Conversion Programs Using A Data Definition Language," Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, 1973.

[RCA 73] RCA, "Programming Manual For Electronic Quality Assurance Test Equipment, Automatic (EQUATE)," Developed For U.S. Army, May 1973.

[RIN 76] Rin, N.A., "Automatic Generation of Business Data Processing Programs From A Non-procedural Language," Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, 1976.

[RYT 76] Ruth, G., "Protosystem I: An Automatic Programming System Prototype," Laboratory for Computer Science, MIT, June 1976.

[SEM 74] "Digest of Papers: 1974 Semiconductor Test Symposium," 74CH0909-2C, IEEE Society, November 1974.

[SHA 76] Shastry, S.K., "Interactive Automatic Programming For Data Processing," Dissertation Proposal in Computer and Information Science, University of Pennsylvania, July 1976.

[STE 74] Stevens, Myers and Constantine, "Structural Design," IBM Systems Journal, 13-2, 1976, pp. 115-139.

[TEI 71] Teichroew, D. and Sayani, H., "Automation of Systems Building,: Datamation, August 1971, pp. 25-30.

[TEI 74] Teichroew, D., Hersley, E., and Bastarache, M., "An Introduction to PSL/PSA," ISDOS Working Paper No. 86, Dept. of Industrial and Operations Engineering, University of Michigan, March 1974.

[TEI 76] Teichroew, D., "ISDOS and Recent Extensions," Proceedings of the Symposium on Computer Software Engineering," April 1976.

[THA 71] Thall, R.M., "A Manual for PSA/ADS: A Machine-Aided Approach to Analysis of ADS," ISDOS Working Paper No. 35, Dept. of Industrial Engineering, University of Michigan, Ann Arbor, 1971.

[TIN 77] Tinaztepe, C., "Automatic Test Design," Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, 1977.

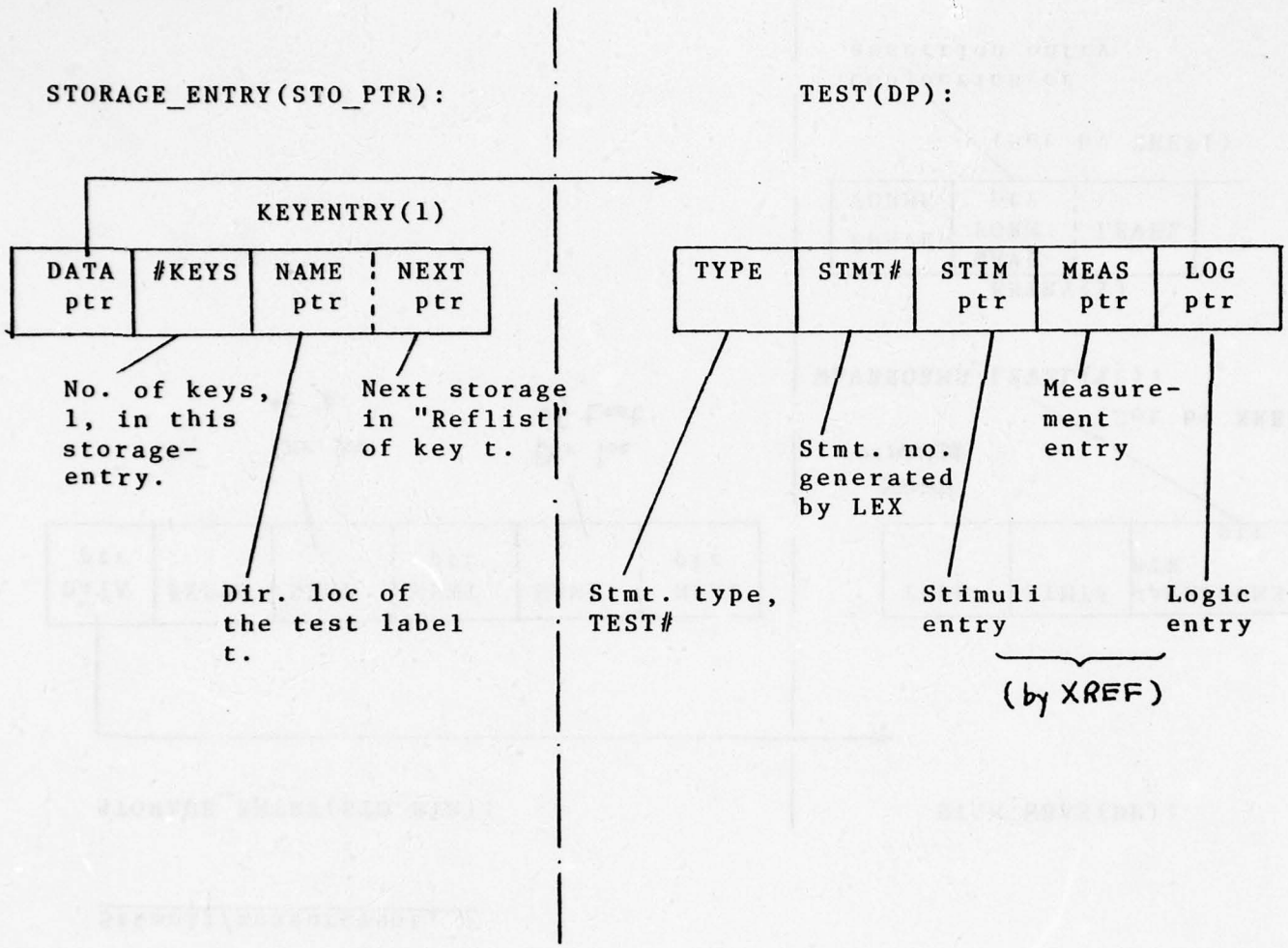
[TO 74] To K. and Tulloss, R., "Automatic Test Systems," IEEE Spectrum, September 1974, pp. 44-52.

[WAR 74] Warshall, S., et.al., "CTL-Compass Test Language," Frankford Arsenal Report R-3017, MCA, June 1974.

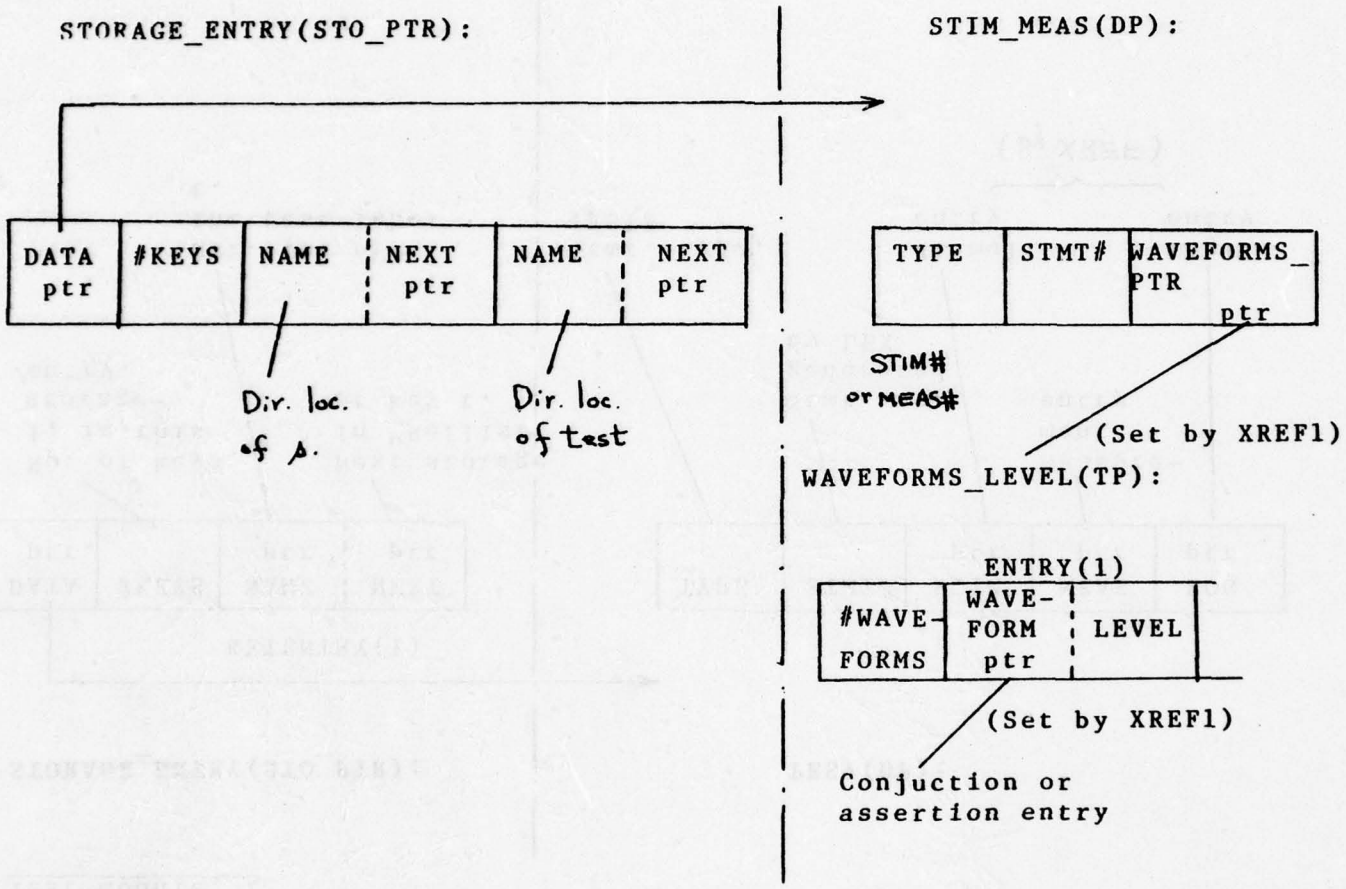
APPENDIX A

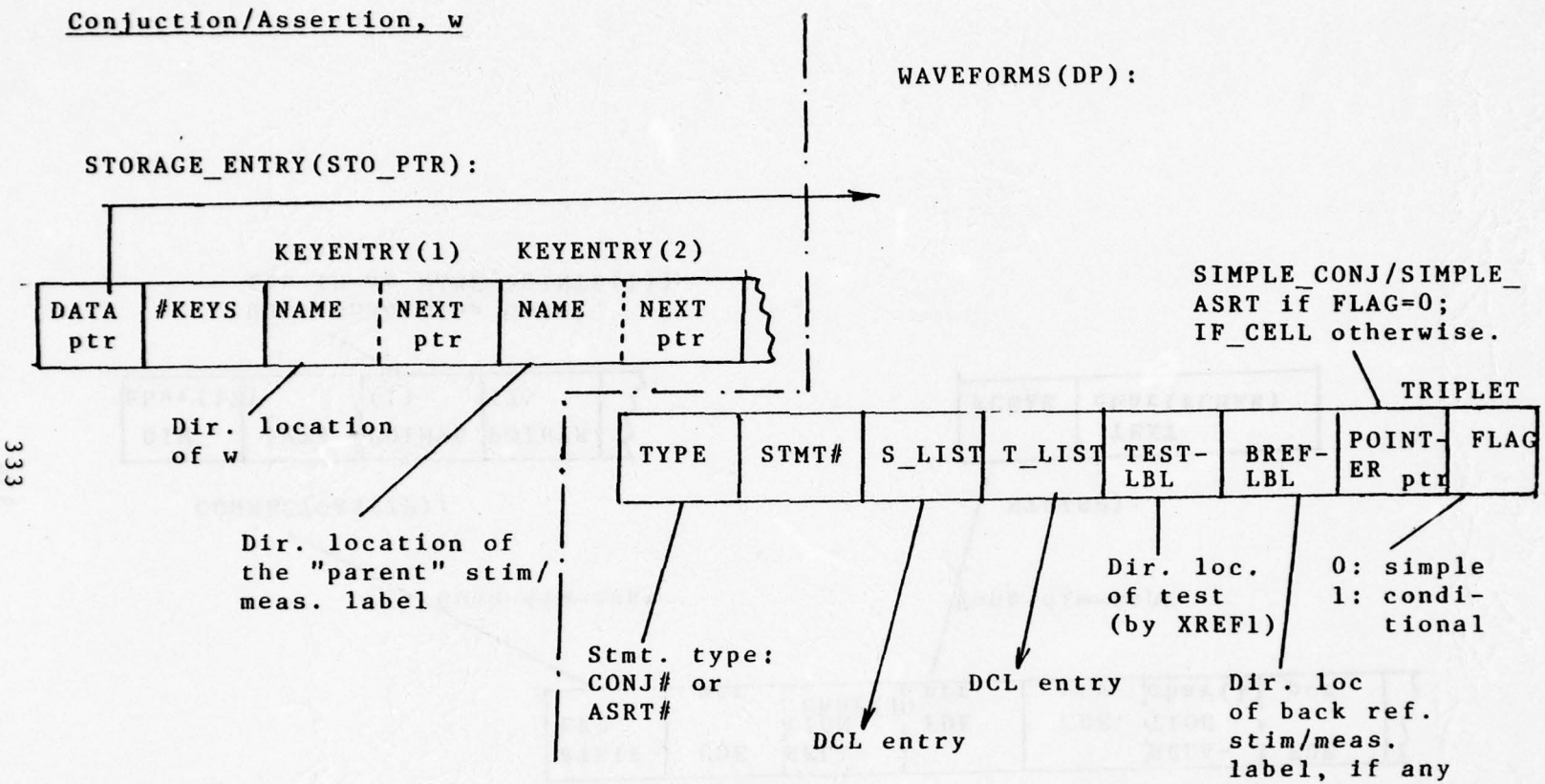
NOPAL SYSTEM DATA STRUCTURES

This appendix illustrates the data structures used to encode all of the sixteen types of statements (as listed in Table 5.9 of Chapter 5) in the NOPAL system. Associated with each type of statement are two data structures: (1) storage entry and (2) actual data. Each data structure is implemented by a PL/1 BASED structure IBM 72 and is identified by its main (i.e., level 1) structure name followed by its based pointer enclosed in parentheses. In the drawing, each cell ("square") represents a field in the data structure. The attribute of a field is normally integer (i.e. FIXED BINARY) unless specified otherwise.

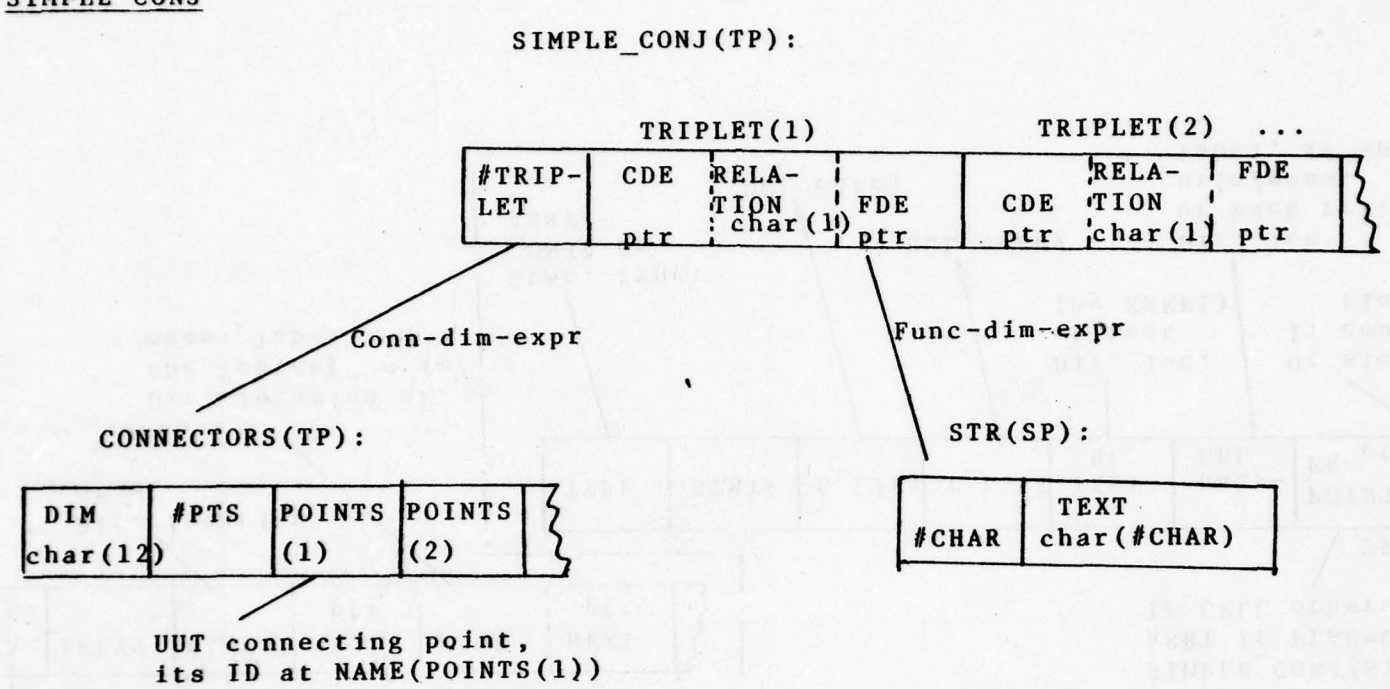

Test module, t
STORAGE_ENTRY(STO_PTR):
TEST(DP):
KEYENTRY(1)
DATA ptr
#KEYS
NAME ptr
NEXT ptr
TYPE
STMT#
STIM ptr
MEAS ptr
LOG ptr
No. of keys, 1, in this storage-entry.
Next storage in "Reflist" of key t.
Dir. loc of the test label t.
Stmt. type, TEST#
Stmt. no. generated by LEX
Measure-ment entry
Stimuli entry
Logic entry
(by XREF)

Stimuli/measurement, s



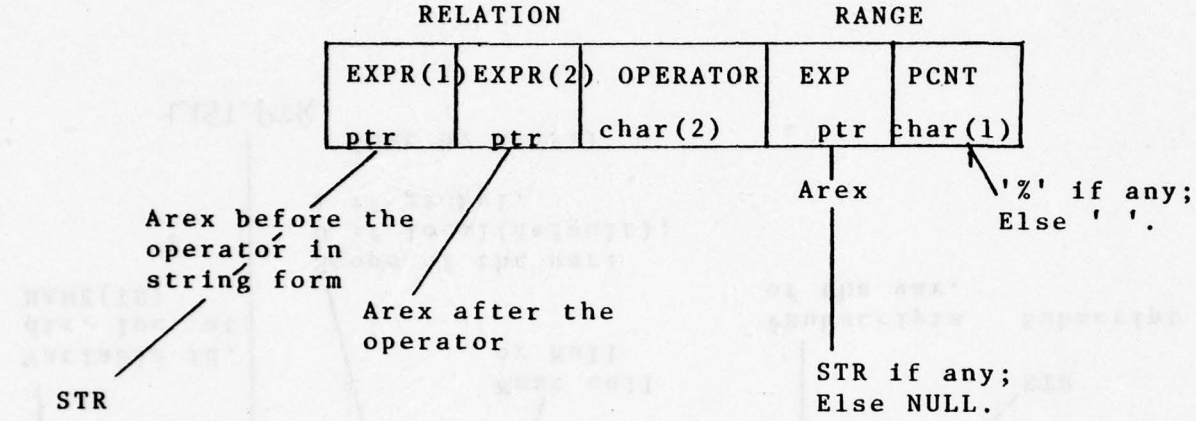

Conjuction/Assertion, w
WAVEFORMS(DP):
STORAGE_ENTRY(STO_PTR):
KEYENTRY(1)
KEYENTRY(2)
DATA ptr
#KEYS
NAME
NEXT ptr
NAME
NEXT ptr
SIMPLE_CONJ/SIMPLE_ASRT if FLAG=0; IF_CELL otherwise.
TRIPLET
TYPE
STMT#
S_LIST
T_LIST
TEST-LBL
BREF-LBL
POINT-ER ptr
FLAG
Dir. location of w
Dir. location of the "parent" stim/meas. label
Stmt. type: CONJ# or ASRT#
DCL entry
DCL entry
Dir. loc. of test (by XREF1)
Dir. loc of back ref. stim/meas. label, if any
0: simple
1: condi-tional

SIMPLE CONJ


SIMPLE_CONJ(TP):
TRIPLET(1)
TRIPLET(2) ...
#TRIP-LET
CDE ptr
RELA-TION char(1)
FDE ptr
CDE ptr
RELA-TION char(1)
FDE ptr
Conn-dim-expr
Func-dim-expr
CONNECTORS(TP):
STR(SP):
DIM char(12)
#PTS
POINTS (1)
POINTS (2)
#CHAR
TEXT char(#CHAR)
UUT connecting point,
its ID at NAME(POINTS(1))

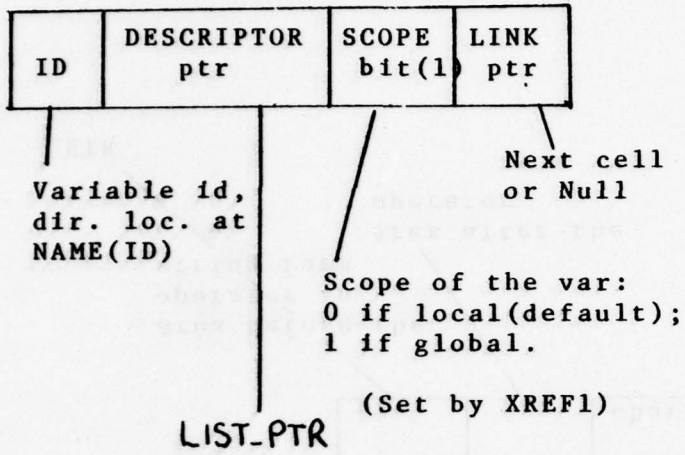
SIMPLE-ASRT

SIMPLE_ASRT(TP):

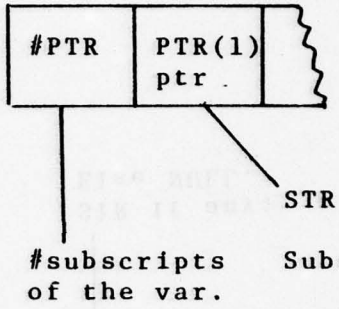


DECLARATION (DCL)

DCL(TP):

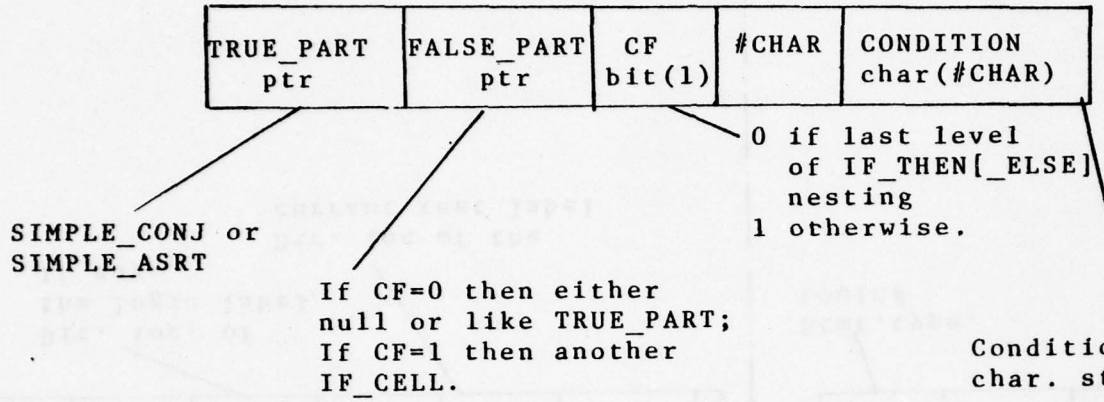


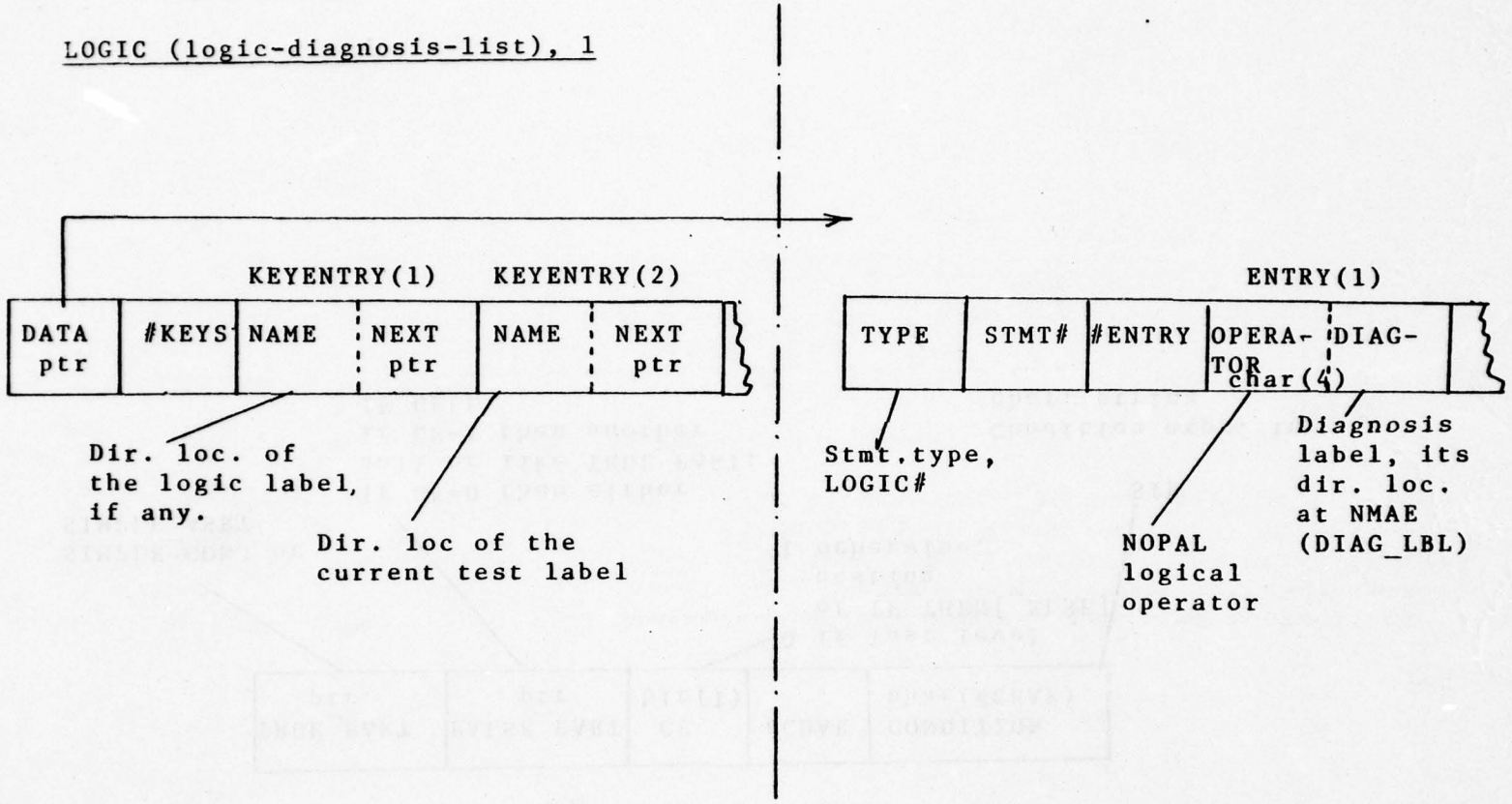
LIST_PTR(TP):



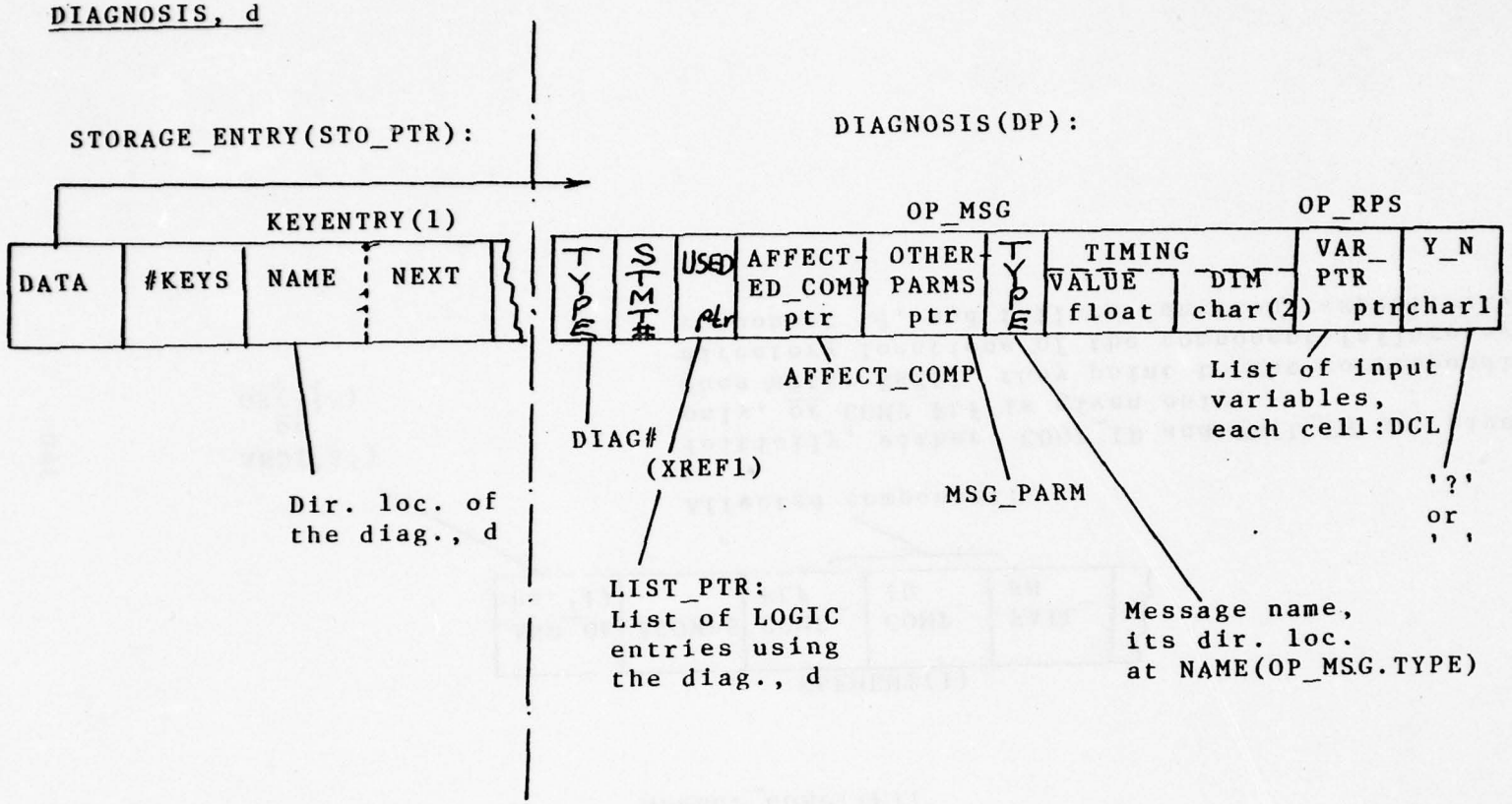
IF-CELL

IF_CELL(TP):



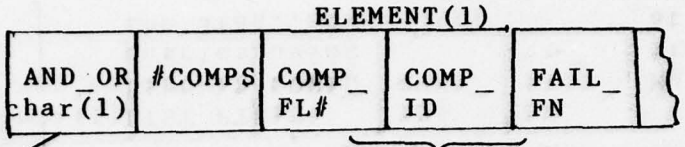

LOGIC (logic-diagnosis-list), 1
KEYENTRY(1)
KEYENTRY(2)
ENTRY(1)
DATA ptr
#KEYS
NAME
NEXT ptr
NAME
NEXT ptr
TYPE
STMT#
#ENTRY
OPERA-TOR
DIAG-
char(4)
Dir. loc. of the logic label, if any.
Dir. loc of the current test label
Stmt.type, LOGIC#
NOPAL logical operator
Diagnosis label, its dir. loc. at NMAE (DIAG_LBL)

DIAGNOSIS, d


STORAGE_ENTRY(STO_PTR):
DIAGNOSIS(DP):
KEYENTRY(1)
OP_MSG
OP_RPS
DATA
#KEYS
NAME
NEXT
TYPE
STMT#
USED ptr
AFFECTED_COMP ptr
OTHER PARMS ptr
TYPE
TIMING
VALUE float
DIM char(2)
VAR_PTR ptr
Y_N char1
AFFECT_COMP
List of input variables, each cell:DCL
DIAG#
(XREF1)
MSG_PARM
'?' or ' '
Dir. loc. of the diag., d
LIST_PTR: List of LOGIC entries using the diag., d
Message name, its dir. loc. at NAME(OP_MSG.TYPE)

AFFECTED COMPONENTS

AFFECT_COMP(TP):



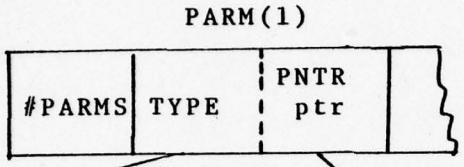
Affected component:

AND('&')
or
OR('|')

Initially, either COMP_ID and FAIL_FN are given only, or COMP_FL# is given only.
Then after XREF1, they point to the corresponding directory locations of the component-failure-seq# component id, and failure function, respectively.

OP-MSG PARAMETERS

```
MSG_PARM(TP);
```

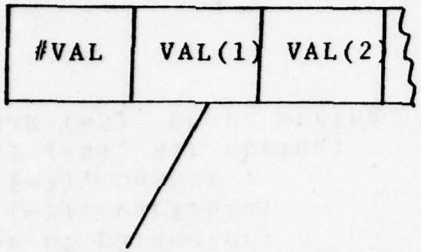


Type of parameter:
$A (=2), variable;
$D (=3), number ;
$BIT (=4), Bit string;
$CHAR (=5), Char. string.

STR, if TYPE≠$A;
DCL, if TYPE=$A.

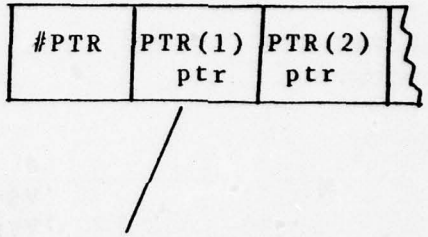
LIST-VAL/LIST-PTR (list of values or pointers)

LIST_VAL(TP):



Integer, FIXED BIN(15,0)

LIST_PTR(TP):



PL/I pointer, e.g. points to a BASED structure

MESSAGE, m

STORAGE_ENTRY(STO_PTR):

KEYENTRY(1)

DATA ptr	#KEYS	NAME	NEXT	

Dir. loc. of the message m.

MESSAGE(DP):

TYPE	STMT#	ALIAS	#ELEM	TXTELEM(1) ptr	

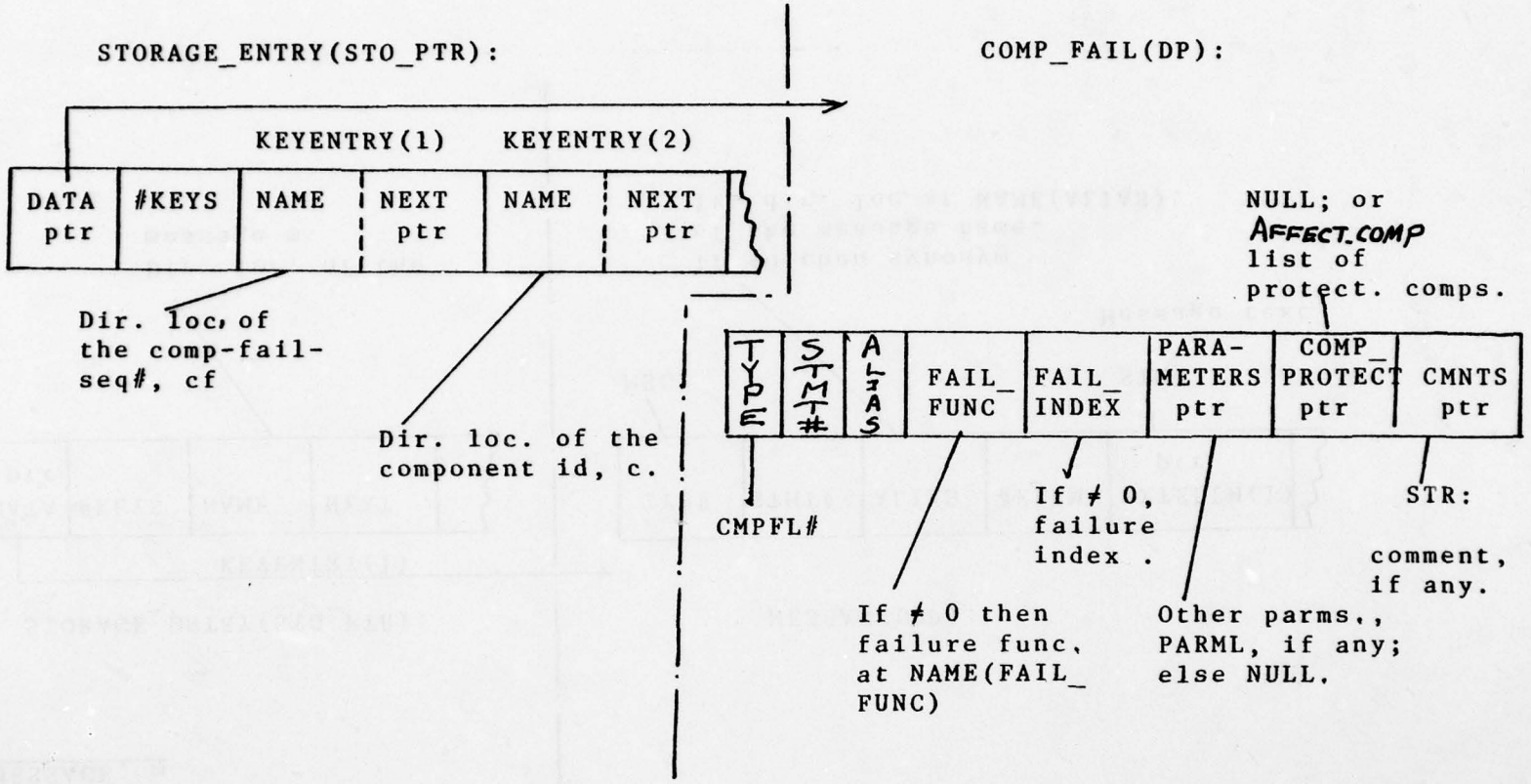
MSG#

STR:

Message text

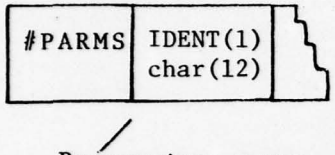
If ≠0 then synosym of the message name. Its dir. loc. at NAME(ALIAS).

COMPONENT FAILURE, cf: f(c)

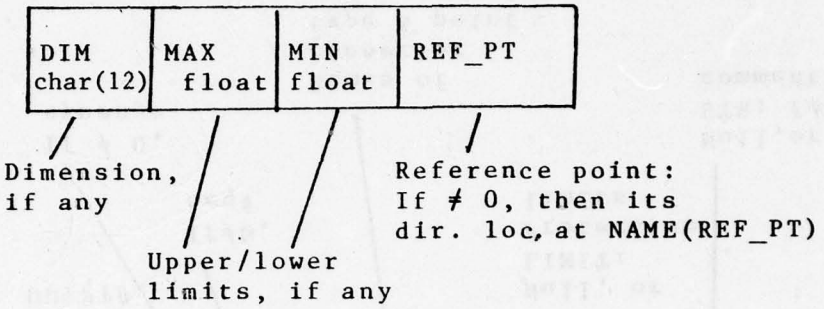


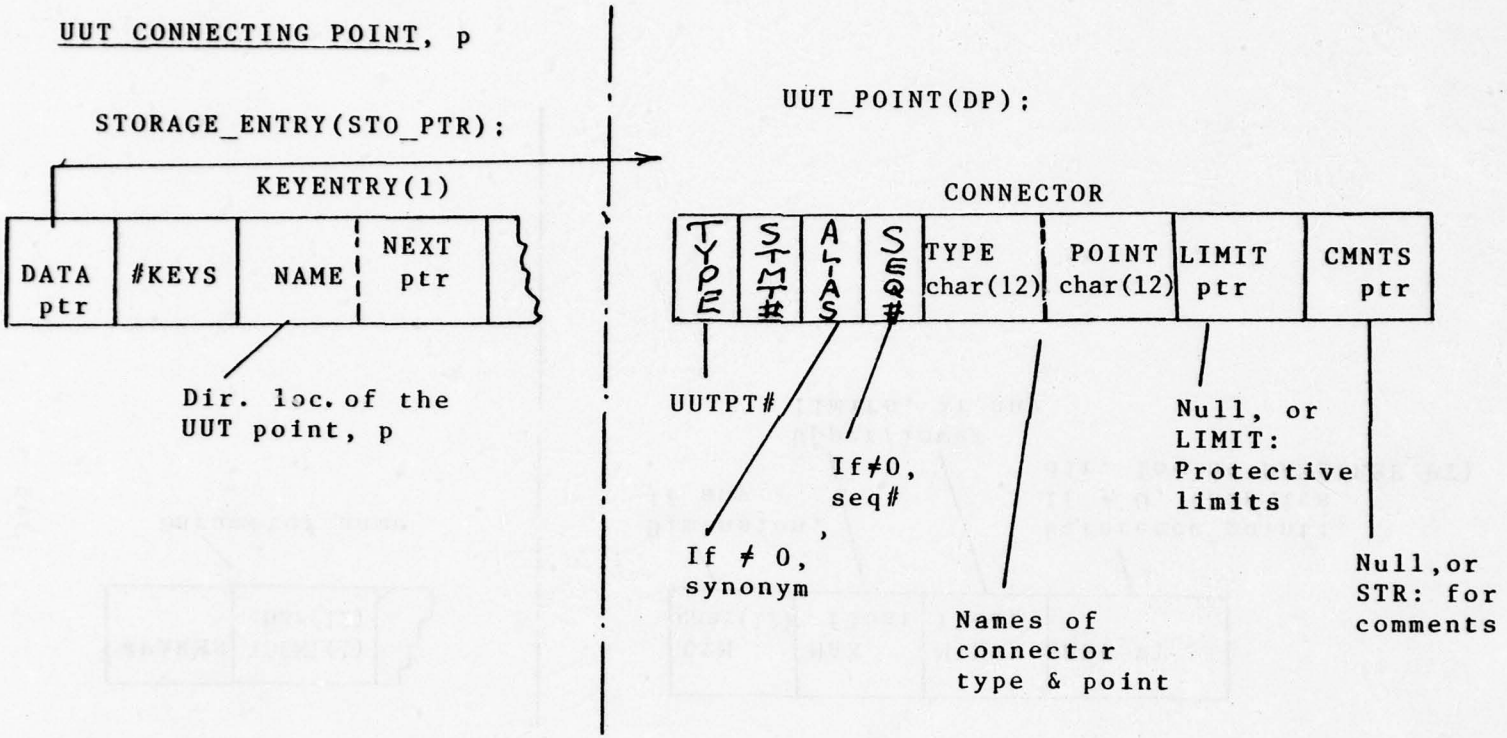
PARAMETER LIST/PROTECTIVE LIMITS

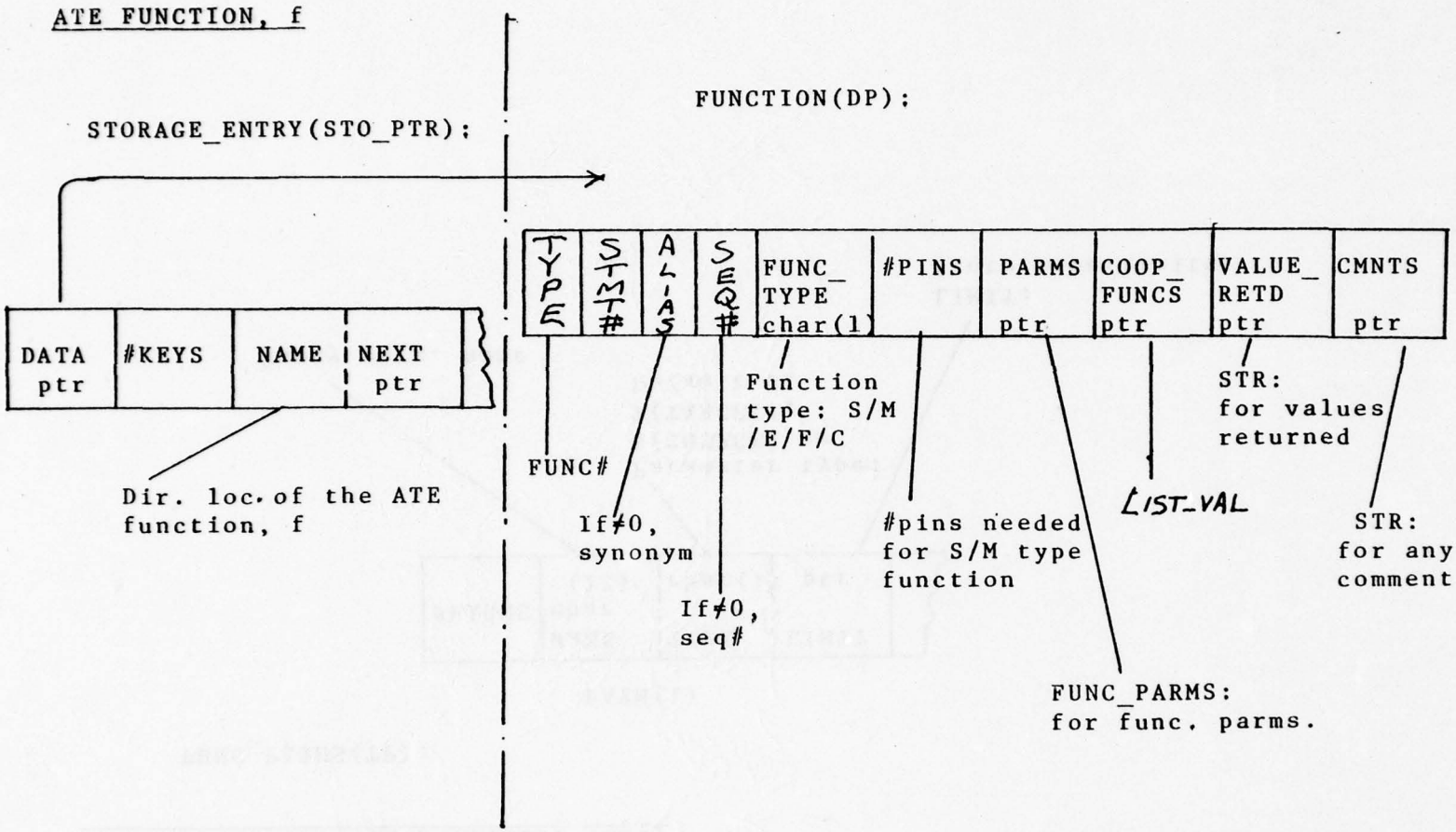
PARAMETER IIST/
PARML(TP):



LIMIT(TP):

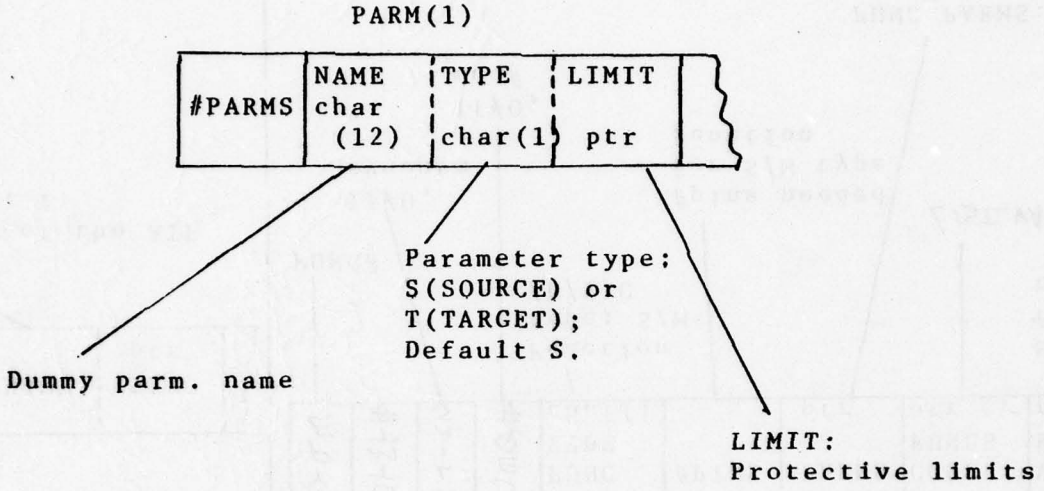


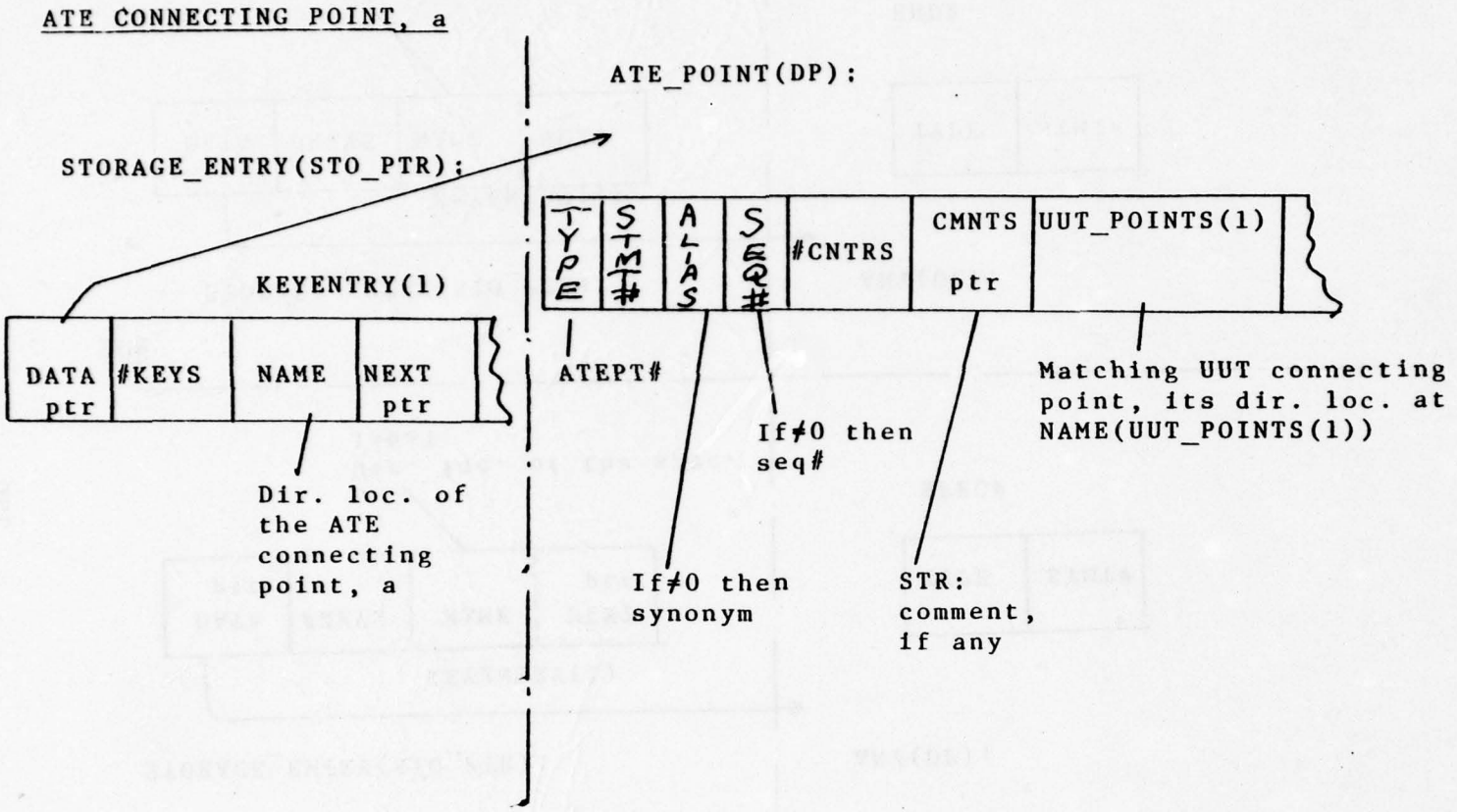

UUT CONNECTING POINT, p
STORAGE_ENTRY(STO_PTR);
KEYENTRY(1)
DATA ptr
#KEYS
NAME
NEXT ptr
Dir. loc. of the UUT point, p
UUT_POINT(DP);
CONNECTOR
TYPE
STMT#
ALIAS
SEQ#
TYPE char(12)
POINT char(12)
LIMIT ptr
CMNTS ptr
UUTPT#
If ≠ 0, synonym
If≠0, seq#
Names of connector type & point
Null, or LIMIT: Protective limits
Null, or STR: for comments


ATE FUNCTION, f
STORAGE_ENTRY(STO_PTR);
FUNCTION(DP);
DATA ptr
#KEYS
NAME
NEXT ptr
TYPE
STMT#
ALIAS
SEQ#
FUNC_ TYPE char(1)
#PINS
PARMS ptr
COOP_ FUNCS ptr
VALUE_ RETD ptr
CMNTS ptr
Dir. loc. of the ATE function, f
FUNC#
If≠0, synonym
If≠0, seq#
Function type: S/M /E/F/C
#pins needed for S/M type function
FUNC_PARMS: for func. parms.
LIST_VAL
STR: for values returned
STR: for any comment

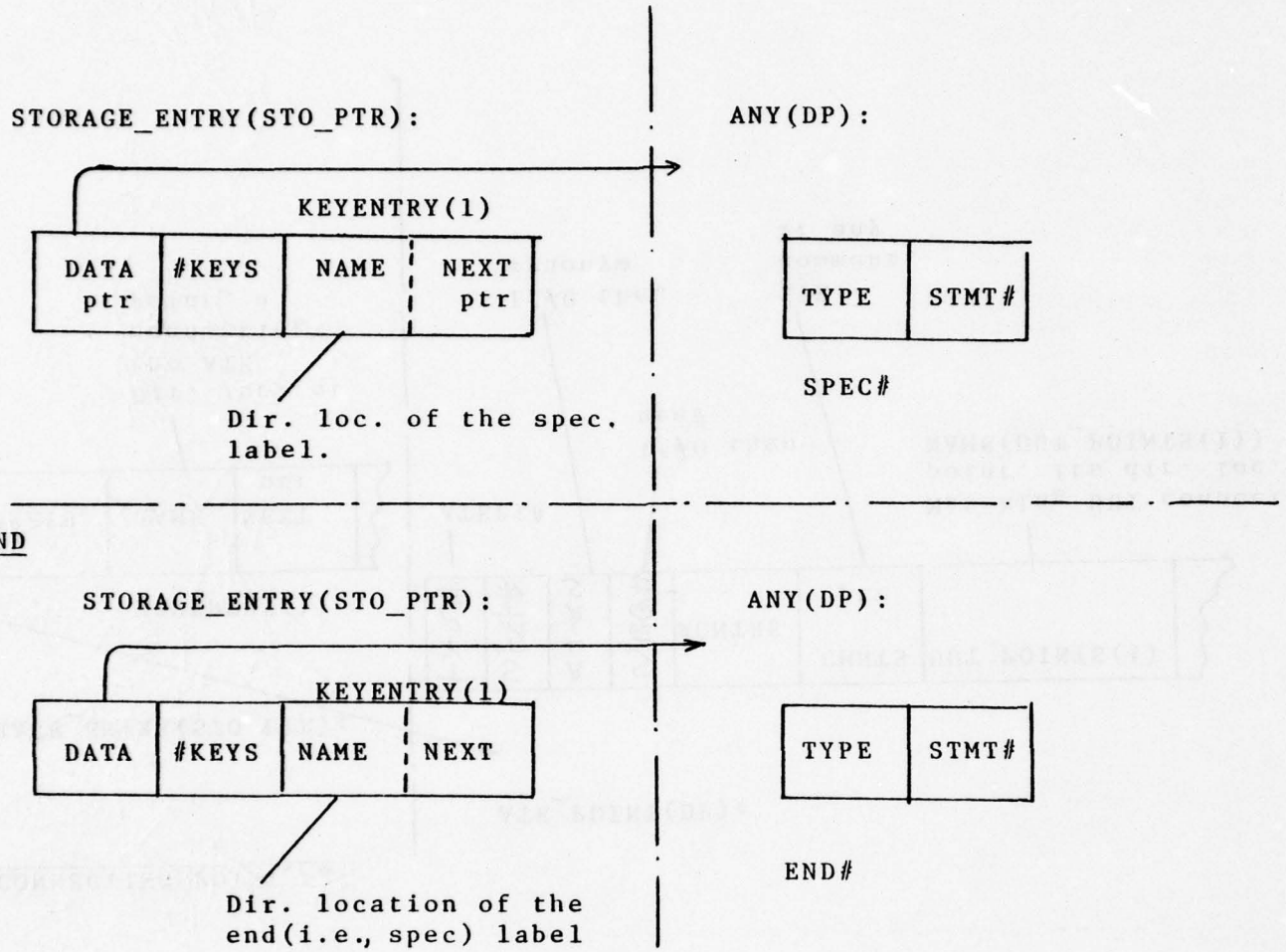
FUNC-PARMS(Function parameter list)

FUNC_PARMS(TP):




ATE CONNECTING POINT, a
ATE_POINT(DP):
STORAGE_ENTRY(STO_PTR):
KEYENTRY(1)
DATA ptr
#KEYS
NAME
NEXT ptr
TYPE
STMT#
ALIAS
SEQ#
#CNTRS
CMNTS ptr
UUT_POINTS(1)
ATEPT#
If≠0 then seq#
Matching UUT connecting point, its dir. loc. at NAME(UUT_POINTS(1))
Dir. loc. of the ATE connecting point, a
If≠0 then synonym
STR: comment, if any

SPECIFICATION



APPENDIX B

AN EXAMPLE OF USING THE NOPAL LANGUAGE AND PROCESSOR

This appendix provides a complete example of the use of the NOPAL language and Processor. The sample problem is the MINIRADIOSET test specification of Figure 3.2.

Included in the appendix are job control statements used to invoke the NOPAL Processor, source and reformatted specification listings, cross reference reports, and the precedence matrix and processing sequence of each test module.

```
//CHANGFEB JOB (012499,####,CCCC,#10$2,2),CHANG,REGION=384K                    JOB 1375
***ROUTE PRINT LOCAL
***PROCLIB U.P012499.PROCLIB
//  EXEC NOPAL,TIME=(,10)
XXNOPAL  PROC  O='U.P012499',D=KEEP    "NOPAL"    SET D=DELETE TO KILL      00000100
***    RUN "NOPAL", USING LOAD-MODULE LIB "PGMLIB"                           00000200
XXGO    EXEC PGM=NOPAL,REGION=384K,TIME=(,15),                               00000300
XX     PARM='SAPIN=(1,80,80),SAPLIST,XREF1,SOURCE2,XREF2,SEQ=1,NOCODE'       00000400
XXSTEPLIB  DD DSN=&Q..PGMLIB,DISP=(OLD,&D)    "NOPAL PROGRAM LIB"           00000500
IEF653I SUBSTITUTION JCL - DSN=U.P012499.PGMLIB,DISP=(OLD,KEEP)
XXSAPLIST  DD SYSOUT=A                          "SOURCE LISTING FILE"        00000600
XXSAPERR   DD SYSOUT=A                          "SYNTAX ERRORS FILE"         00000700
XXXREF1  DD SYSOUT=A                            "XREF/ATTR"                  00000800
XXXREFERR DD SYSOUT=A                           "XREF/ATTR ERRORS/WARNINGS"  00000900
XXSOURCE2  DD SYSOUT=A                          "SUMMSRY SOURCE LISTINS"     00001000
XXXREF2  DD SYSOUT=A                            "SUMMSRY CROSS-REF."         00001100
XXSAPIN  DD DDNAME=SYSIN                        "NOPAL TEST SPEC"            00001200
XXSEQRPT DD SYSOUT=A                            "SEQENCE/ANALYSIS REPORTS"   00001220
XXSEQERR DD SYSOUT=A                            "SEQENCE/ANALYSIS ERRORS"    00001240
XXSYSPRINT DD SYSOUT=A                          "OTHER SYSTEM MESSAGE"       00001260
***ENDNOPAL PEND     "END OF RUN NOPAL"                                      00001300
//SYSIN  DD *
//
IEF236I ALLOC. FOR CHANGFEB GO
IEF237I 13C   ALLOCATED TO STEPLIB
IEF237I A59   ALLOCATED TO SAPLIST
IEF237I A5A   ALLOCATED TO SAPERR
IEF237I A5B   ALLOCATED TO XREF1
IEF237I A5C   ALLOCATED TO XREFERR
IEF237I A5D   ALLOCATED TO SOURCE2
IEF237I A5E   ALLOCATED TO XREF2
IEF237I AA9   ALLOCATED TO SAPIN
IEF237I A5F   ALLOCATED TO SEQRPT
IEF237I A60   ALLOCATED TO SEQERR
IEF237I A61   ALLOCATED TO SYSPRINT
IEF142I - STEP WAS EXECUTED - COND CODE 0000
IEF285I   U.P012499.PGMLIB                                KEPT
IEF285I   VOL SER NOS= UPDAS9.
IEF373I STEP /GO      / START 77084.1437
IEF374I STEP /GO      / STOP  77084.1439 CPU    0MIN 03.04SEC STOR VIRT 216K
IEF375I  JOB /CHANGEEB/ START 77084.1437
IEF376I  JOB /CHANGEEB/ STOP  77084.1439 CPU    0MIN 03.04SEC
```

```
NOPAL PROCESSOR OPTIONS SPECIFIED: SAPIN=(1,80,80),SAPLIST,XREF1,SOURCE2,XREF2,SEQ=1,NOCODE

STMT NO.
--------

    1       SPEC MINIRADIOSET;

              /*  TEST  MODULES  */

    2         TEST OC_INPUT;   /* OC INPUT SHORT CHECK */
              /* NO STIMULI */
    3           MEASUREMENT;
    4             CONJUNCTION:  <J24_A, J24_C> = CONST_R(MRES OHM)
    4                           TARGET:  MRES;
    5             ASSERTION:  MRES > 100      SOURCE:  MRES;
    6           LOGIC:  |¬ D2, *D3;
    7               DIAGNOSIS D2: (INPUT_SHORT,, #4);
    8               DIAGNOSIS D3:  (, (MRES, ' OHMS'),D);

    9         TEST AMPL;
   10           MEAS;
   11             ASRT:   V1 = 0.26 +- 0.06;
   12           LOGIC:  *D7, |¬D8;
   13               DIAG D7:  PARM=(V1, 'VRMS'), TYPE= D;
   14               DIAG D8:  AFFECTED COMP=AMPL_TOL(STD_5MHZ_FREQ), TYPE= #6,
   14                         OTHER PARM= 'AMPL';

   15         TEST DISTORT_2W;
   16           STIM;
   17             CONJ:  SAME AS OCV_AMS;
   18           MEAS;
   19             CONJ:  <J19_L, GND> = DISTORTION(P1%, 1 KHZ)
   19                          TARG:  P1;
   20             ASRT:  P1 <= 5;
   21           LOGIC:  *27, |¬30;
   22               DIAG 30:  PARAM= (' 2W', 5.0), TYPE= #18, COMP=DISTORT(AUDIO_2W);

   23         TEST DISTORT_VOLT;
   24           STIM  OCV_AMS;
   25             CONJ:  SAME AS OCV EXCEPT
   25                         J16 = SIGNAL_AM(25.002MHZ, +13DB, 0%, 1KHZ);
   26           MEAS;
   27             CONJ: <J19_A, GND> = SINE_D(V2 VOLT, *, 0 SEC)
   27                          TARGET: V2;
   28               ASSF  A1:  V2 >= 2.2;
   29               ASRT  A2:  V2 <= 2.8;
   30           LOGIC   *24, *25, |¬26;
   31               DIAG 24:  TYPE=#15, TIME=0, RESPONSE=?;
   32               DIAG 25:  PARMS=(V2, ' VAC'), TYPE = D;
   33               DIAG 26:  TYPE=#17, COMP= REF_VOLT(AUDIO_10MW);

   34         TEST FREQ;
   35           STIM  OCV:   /* 27.5V DC TO PIN J24_B */
   36             CONJ: <J24_B, GND> =  CONST_S(27.5 VOLT);
   37           MEAS;
   38             CONJ:  <J22, GND> = SINE_D(V1 VOLT, F1 HZ, VAR1 SEC)
   38                          TARGET:  V1, F1;
   39             ASRT:  IF VAR1 = 60 THEN  F1 = 5E6 +- 60
```

```
                                    ELSE F1 = 5E6 +- 2.5;
        LOGIC:  *D4, |¬D5, *D6;
            DIAG  D4:  (,, #5, 0), VAR1;
            DIAG  D5:  (FREQ_TOL(STD_5MHZ_FREQ), 'FREQ', #6);
            DIAG  D6:  (, (F1, ' HZ'),0);

      TEST DISTORT_10MW;
        STIM:  SAME AS DCV_AMS;
        MEAS:
          CONJ: <J19_A, GND> = DISTORTION(P1 %, 2 KHZ)
                    TARGET: P1;
          ASRT:  P1 <= 3;
        LOGIC: *27, |¬28;
            DIAG 27:  PARAMETERS=(P1, '%'), TYPE=0;
            DIAG 29:  COMP=DISTORT(AUDIO_10MW),TYPE=#18,
                      PARMS=(' 10MW', 1.0);

       /*  MESSGAE DEFINITIONS  */
      MESSAGE 0:  ALIAS=DISLPAY, TEXT='$T: $P ';
      MESS #4: '$/T DC INPUT SHORTED J24-B/J24-C ',
               'AN/GRC-106 DEFECTIVE.  CHECK PRINTOUTS FOR DEFECTS.',
               'PRESS STOP.';
      MESS #5: 'IF A 12 MINUTE WARMUP IS DESIRED, KEY IN 720;'
               ' OTHERWISE, KEY IN 60.   PRESS YES.';
      MESS  #6:   '$C DEFECTIVE.',
                  '5.0 MHZ STD. OUT OF $P TOLERANCE.' ;
      MESS  #15:  'MC & KC CONTROLS TO 250000.',
                  'ADJUST AUDIO GAIN CONTROL FOR 2.2 TO 2.8 VAC',
                  '(2.5 VAC NOMINAL).    PRESS  YES.';
      MESS  #17:  '10 MW DISTORTION REFERENCE VOLTTAGE FAILED.';
      MESS  #18:  'SPL AUDIO DISTORTION GREATER THAN $P2 PERCENT.';

       /*  UUT CONNECTING POINTS  */
      UUT_POINT:  J22, LIMIT=(VOLT, 70, 0, GND);
      UUT_PT 2:  J24_B, ALIAS = XJ24_B, CONNECTOR=(MULTIPLE, B),
                 LIMIT=(VOLT, 35, 20, GND);
      UUT_P:   J24_C, ALIAS=GND, CONN=(MULTIPLE, C);
      UUT_PT:    J16, CONNECTOR=COAXIAL, LIMIT=(UVOLT, 100, 0, GND),
                      ' COAXIAL CABLE';
      UUT_PT:   J19_A, LIMIT=(VOLT, 5, 0, GND);
      UUT_PT:   J19_I, LIMIT=(VOLT, 70, 0, GND);
      UUT_PT:    J19_B, ALIAS=GND;

       /*  UUT COMPONENT/FAILURES  */
      COMPONENT  1:   INPUT_SHORT;
      COMP_FAIL  2:  STD_5MHZ_FREQ, FAILURE FUNCTION= FREQ_TOL,
                     INDEX= 1, PROTECTION = 1;
      COMP_FAIL 3:   STD_5MHZ_FREQ, FAIL FUNC = AMPL_TOL, INDEX = 2,
                     PROTECT = 1;
      COMP  6:  AUDIO_10MW, FAIL = REF_VOLT, PROT=1, 'DISTORTION REF VOLT';
      COMP_FL  7:  AUDIO_10MW, FAIL = DISTORT, PROTECTION=(1, 6);
      COMP_FL:   AUDIO_2M, FAIL FUNC= DISTORT;

       /*     ATE FUNCTIONS */
      FUNC 10:  CONST_S, FUNCTION TYPE =S, PARM= (X,S,LIMIT=(VOLT,60,0)),
                      VALUE RETURNED =' CONSTANT VOLT.';
      FUNC 20:  CONST_P, TYPE=M, PARAM= (X,T,(OHM,1000,1)),  VALUE='TRUE/FALSE';
      FUNC 30:  SINE_D,ALIAS=SINE_DELAY, TYPE=M, PARM1= (X,T,(VOLT,10,-10)),
          PARM2=(Y,D,(MHZ,10,0)), PARM2=(Z,S,SEC), 'AMPL., FREQ., TIME DELYD';
      FUNCTION  40: DISTORTION, TYPE=M, PARM= (X,T,Z),  /* % DIST. */
                PARM=(Y,S,(KHZ,100,0))  /* FREQ.  */,    VALUE='TRUE/FALSE';
      FUNCTION 50: SIGNAL_AM, ALIAS=SAM, TYPE=S, *PINS = 1,
                PARM#1= (X,S,(MV,100,0.1)),   /* (X,Y,Z) FREQ */

                PARM#2= (Y,S,(DB,-10,-150) ),  /*   POWER */
                PARM#3= (Z, S, %),    /* MODULATION IN PERCENT  */
                PARM#4= (W, S, (KHZ,15,0.1))  /*  MOD FREQ.  */;
      FUNC 110:  FREQ_TOL, TYPE=F, PARM=COMPONENT;
      FUNC 120:  AMPL_TOL, TYPE=F, PARM=COMPONENT;
      FUNC 130:  REF_VOLT, TYPE=F, PARM=COMPONENT;
      FUNC 140: DISTORT, TYPE=F, PARM=COMPONENT;

       /*  ATE INTER-CONNECTING POINTS  */
      ATE_POINT:  ATE_J24B, UUT_PT=J24_B;

    END MINIRADIOSET;
```

```
ERROR/WARNING MESSAGES GENERATED DURING NOPAL SYNTAX ANALYSIS:

*WARNING*   IN STATEMENT    14, NEAR TEXT 'STD_5MHZ_FRE', NAME/INTEGER WAS TOO LONG.   TRUNCATED.
*WARNING*   IN STATEMENT    42, NEAR TEXT 'STD_5MHZ_FRE', NAME/INTEGER WAS TOO LONG.   TRUNCATED.
*WARNING*   IN STATEMENT    67, NEAR TEXT 'STD_5MHZ_FRE', NAME/INTEGER WAS TOO LONG.   TRUNCATED.
*WARNING*   IN STATEMENT    68, NEAR TEXT 'STD_5MHZ_FRE', NAME/INTEGER WAS TOO LONG.   TRUNCATED.

*STATISTICS*  NO. OF SAP ERRORS =     0,     NO. OF WARNINGS =     4,     NO. OF STATEMENTS =    82
```

CROSS REFERENCE AND ATTRIBUTES REPORT

NAME	DEF NO.	ATTRIBUTES AND REFERENCES
#15	56	MESSAGE LABEL 31
#17	57	MESSAGE LABEL 33
#18	58	MESSAGE LABEL 22 51
#4	53	MESSAGE LABEL 7
#5	54	MESSAGE LABEL 41
#6	55	MESSAGE LABEL 14 42
AMPL	9	TEST LABEL 10 12
AMPL_TOL	78	ATE-FUNCTION ID, F 14 68
ATE_J24B	81	ATE-POINT ID
AUDIO_10MW	69	COMPONENT ID, WITH FAILURE-FUNCTION: REF_VOLT 33
AUDIO_10MW	70	COMPONENT ID, WITH FAILURE-FUNCTION: DISTORT 51
AUDIO_2W	71	COMPONENT ID, WITH FAILURE-FUNCTION: DISTORT 22
A1	28	ASSERTION LABEL
A2	29	ASSERTION LABEL
CONST_R	73	ATE-FUNCTION ID, M 4
CONST_S	72	ATE-FUNCTION ID, S 36 25 17 45
D	52	MESSAGE LABEL 8 13 32 43 50
DC_INPUT	2	TEST LABEL 3 6
DCV	35	STIMULUS LABEL 36 25
DCV_AMS	24	STIMULUS LABEL 25 17 45
DISLPAY	52	SYNONYM OF MESSAGE LABEL: D
DISTORT	80	ATE-FUNCTION ID, F 22 51 70 71
DISTORT_VOLT	23	TEST LABEL 24 26 30
DISTORT_10MW	44	TEST LABEL 45 46 49
DISTORT_2W	15	TEST LABEL 16 18 21
DISTORTION	75	ATE-FUNCTION ID, M 19 47
D2	7	DIAGNOSIS LABEL 6
D3	8	DIAGNOSIS LABEL 6
D4	41	DIAGNOSIS LABEL 40
D5	42	DIAGNOSIS LABEL 40

```
D6                  43      DIAGNOSIS LABEL
                              40
D7                  13      DIAGNOSIS LABEL
                              12
D8                  14      DIAGNOSIS LABEL
                              12
FREQ                34      TEST LABEL
                              35  37  40
FREQ_TOL            77      ATE-FUNCTION ID, F
                              42  67
F1                  38      VARIABLE ID
                              39  43
GND                 65      SYNONYM OF OUT-POINT ID: J24_C, J19_B
                              19  27  36  38  47  59  60  62  63  64  25  17  45
INPUT_SHORT         66      COMPONENT ID
                              7
J16                 62      OUT-POINT ID
                              25  17  45
J19_A               63      OUT-POINT ID
                              27  47
J19_B               65      OUT-POINT ID
J19_L               64      OUT-POINT ID
                              19
J22                 59      OUT-POINT ID
                              38
J24_B               60      OUT-POINT ID
                              4  36  81  25  17  45
J24_C               61      OUT-POINT ID
                              4
MINIRADIOSET         1      SPECIFICATION LABEL
                              82
MRES                 4      VARIABLE ID
                              5  8
P1                  19      VARIABLE ID
                              20  50
P1                  47      VARIABLE ID
                              48  50
REF_VOLT            79      ATE-FUNCTION ID, F
                              33  69
SAM                 76      SYNONYM OF ATE-FUNCTION ID: SIGNAL_AM, S
SIGNAL_AM           76      ATE-FUNCTION ID, S
                              25  17  45
SINE_D              74      ATE-FUNCTION ID, M
                              27  38
SINE_DELAY          74      SYNONYM OF ATE-FUNCTION ID: SINE_D, M
STD_5MHZ_FRE        67      COMPONENT ID, WITH FAILURE-FUNCTION: FREQ_TOL
                              42
STD_5MHZ_FRE        68      COMPONENT ID, WITH FAILURE-FUNCTION: AMPL_TOL
                              14
VAR1                41      VARIABLE ID
                              38  39
V1                  38      VARIABLE ID, GLOBAL
                              13  11
V2                  27      VARIABLE ID
                              28  29  32
XJ24_B              60      SYNONYM OF OUT-POINT ID: J24_B
1                   66      COMPONENT/FAILURE SEQ#
                              67  68  69  70
2                   67      COMPONENT/FAILURE SEQ#
24                  31      DIAGNOSIS LABEL
                              30
25                  32      DIAGNOSIS LABEL
                              30
26                  33      DIAGNOSIS LABEL
                              30
27                  50      DIAGNOSIS LABEL
                              21  49
28                  51      DIAGNOSIS LABEL
                              49
3                   68      COMPONENT/FAILURE SEQ#
30                  27      DIAGNOSIS LABEL
                              21
6                   69      COMPONENT/FAILURE SEQ#
                              70
7                   70      COMPONENT/FAILURE SEQ#
```

ERROR/WARNING MESSAGES GENERATED DURING CROSS-REFERENCE:

STATISTICS NO. OF XREF1 ERRORS = 0 NO. OF WARNINGS = 0

```
/* REFORMATTED SPECIFICATION REPORT, FILE: SOURCE2 */


/*****************************************************/
/*                                                   */
/*  NOPAL TEST SPECIFICATION FOR MINIRADIOSET        */
/*                                                   */
/*****************************************************/

NOPAL SPECIFICATION MINIRADIOSET;

/*****************************************************/
/*                                                   */
/*  TEST MODULES:     6                              */
/*                                                   */
/*****************************************************/


  TEST DC_INPUT;

    /* NULL STIMULI */

    MEASUREMENT $M_DC_INPUT(DC_INPUT);

       CONJUNCTION $M_W0001($M_DC_INPUT):
           (<J24_B, J24_C>  = CONST_R(MRES OHM ))
                   TARGET: MRES;

       ASSERTION $M_W0002($M_DC_INPUT):
           MRES >  100
                   SOURCE: MRES;

    LOGIC $LOGICOO10(DC_INPUT): |¬D2, *D3;

      DIAGNOSIS D2:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=INPUT_SHORT,
              TYPE=#4;

      DIAGNOSIS D3:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(MRES, ' OHMS'),
              TYPE=0;

  TEST AMPL;

    /* NULL STIMULI */

    MEASUREMENT $M_AMPL(AMPL);

       ASSERTION $M_W0001($M_AMPL):
           V1 =  0.26 +- 0.06
                   SOURCE: V1;

    LOGIC $LOGICOO10(AMPL): *D7, |¬D8;

      DIAGNOSIS D7:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(V1, 'VRMS'),
              TYPE=0;

      DIAGNOSIS D8:
```

```
        OPERATOR MESSAGE:
            AFFECTED COMPONENTS=AMPL_TOL(STD_5MHZ_FRF),
            OTHER PARAMETERS=('AMPL'),
            TYPE=#6;

TEST DISTORT_2W;

    STIMULI $S_DISTORT_2(DISTORT_2W);

        CONJUNCTION $S_W0001($S_DISTORT_2):
            (<J24_B, GND>  = CONST_S(27.5 VOLT )) &
            (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

    MEASUREMENT $M_DISTORT_2(DISTORT_2W);

        CONJUNCTION $M_W0001($M_DISTORT_2):
            (<J19_L, GND>  = DISTORTION(P1 % ,1 KHZ ))
                    TARGET: P1;

        ASSERTION $M_W0002($M_DISTORT_2):
            P1 <= 5
                    SOURCE: P1;

    LOGIC $LOGIC0010(DISTORT_2W): *27, |¬30;

      DIAGNOSIS 27:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(P1, '%'),
              TYPE=0;

      DIAGNOSIS 30:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=DISTORT(AUDIO_2W),
              OTHER PARAMETERS=(' 2W', 5.0),
              TYPE=#18;
```

```
            TIME= 0.00000E+00SEC,
        RESPONSE=?;

    DIAGNOSIS 25:
        OPERATOR MESSAGE:
            OTHER PARAMETERS=(V2, ' VAC'),
            TYPE=0;

    DIAGNOSIS 26:
        OPERATOR MESSAGE:
            AFFECTED COMPONENTS=REF_VOLT(AUDIO_10MW),
            TYPE=#17;

TEST FREQ;

  STIMULI DCV(FREQ);

    CONJUNCTION $S_W0001(DCV):
        (<J24_B, GND>  = CONST_S(27.5 VOLT ));

  MEASUREMENT $M_FREQ(FREQ);

    CONJUNCTION $M_W0001($M_FREQ):
        (<J22, GND>  = SINE_D(V1 VOLT ,F1 HZ ,VAR1 SEC ))
                TARGET: F1, V1
                SOURCE: VAR1;

    ASSERTION $M_W0002($M_FREQ):
        IF VAR1=60 THEN
            F1 =  5E+06 +- 60
        ELSE
            F1 =  5E+06 +- 2.5
                SOURCE: VAR1, F1;

  LOGIC $LOGIC0010(FREQ): *D4, |¬D5, *D6;

    DIAGNOSIS D4:
        OPERATOR MESSAGE:
            TYPE=#5,
            TIME= 0.00000E+00SEC,
        RESPONSE=(VAR1);

    DIAGNOSIS D5:
        OPERATOR MESSAGE:
            AFFECTED COMPONENTS=FREQ_TOL(STD_5MHZ_FRE),
            OTHER PARAMETERS=('FREQ'),
            TYPE=#6;

    DIAGNOSIS D6:
        OPERATOR MESSAGE:
            OTHER PARAMETERS=(F1, ' HZ'),
            TYPE=0;
```

```
      CONJUNCTION $M_W0001($M_DISTORT_1):
          (<J19_A, GND>  = DISTORTION(P1 % .2 KHZ ))
                  TARGET: P1;

      ASSERTION $M_W0002($M_DISTORT_1):
          P1 <= 3
                  SOURCE: P1;

    LOGIC $LOGIC0010(DISTORT_10MW): *27, |-28;

     /*** FOLLOWING DIAGNOSIS ALREADY DEFINED BEFORE:

      DIAGNOSIS 27:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(P1, '%'),
              TYPE=0;
                                                          ***/

      DIAGNOSIS 28:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=DISTORT(AUDIO_10MW),
              OTHER PARAMETERS=(' 10MW', 1.0),
              TYPE=#18;

    /*************************************************/
    /*                                               */
    /*  MESSAGES                                     */
    /*                                               */
    /*************************************************/


  MESSAGE #4:
      TEXT='P/T DC INPUT SHORTED J24-B/J24-C ', 'AN/GRC-106 DEFECTIVE.  CHECK P
RINTOUTS FOR DEFECTS.', 'PRESS STOP.';

  MESSAGE 0:  ALIAS=DISPLAY,
      TEXT='$T: $P ';

  MESSAGE #6:
      TEXT='$C DEFECTIVE.', '5.0 MHZ STD. OUT OF $P TOLERANCE.';

  MESSAGE #18:
      TEXT='$P1 AUDIO DISTORTION GREATER THAN $P2 PERCENT.';

  MESSAGE #15:
      TEXT='MC & KC CONTROLS TO 250000.', 'ADJUST AUDIO GAIN CONTROL FOR 2.2 TO
2.8 VAC', '(2.5 VAC NOMINAL).    PRESS  YES.';

  MESSAGE #17:
      TEXT='10 MW DISTORTION REFERENCE VOLTTAGE FAILED.';

  MESSAGE #5:
      TEXT='IF A 12 MINUTE WARMUP IS DESIRED, KEY IN 720:', ' OTHERWISE, KEY IN
60.   PRESS YES.';

    /*************************************************/
    /*                                               */
    /*  UUT COMPONENTS/FAILURES                      */
    /*                                               */
    /*************************************************/


  COMP_FAIL 1: INPUT_SHORT;

  COMP_FAIL 2: STD_5MHZ_FRE, FAILURE FUNCTION=FREQ_TOL, INDEX=1, PROTECT=(1);

  COMP_FAIL 3: STD_5MHZ_FRE, FAILURE FUNCTION=AMPL_TOL, INDEX=2, PROTECT=(1);

  COMP_FAIL 6: AUDIO_10MW, FAILURE FUNCTION=REF_VOLT, PROTECT=(1)
      COMMENTS='DISTORTION REF VOLT';

  COMP_FAIL 7: AUDIO_10MW, FAILURE FUNCTION=DISTORT, PROTECT=(1), 6;

  COMP_FAIL 8&10: AUDIO_2W, FAILURE FUNCTION=DISTORT;

    /*************************************************/
    /*                                               */
    /*  UUT CONNECTION PROMPTS                       */
    /*                                               */
    /*************************************************/
```

```
UUT_POINT      2: J24_B, ALIAS=XJ24_B, CONNECTOR=(MULTIPLE, B),
        LIMIT=(VOLT,  3.50000E+01,  2.00000E+01, GND);

UUT_POINT      : J24_C, ALIAS=GND, CONNECTOR=(MULTIPLE, C);

UUT_POINT      : J19_L, LIMIT=(VOLT,  7.00000E+01,  0.00000E+00, GND);

UUT_POINT      : J16, CONNECTOR=(COAXIAL, ),
        LIMIT=(UVOLT,  1.00000E+02,  0.00000E+00, GND),
        COMMENTS=' COAXIAL CABLE';

UUT_POINT      : J19_A, LIMIT=(VOLT,  5.00000E+00,  0.00000E+00, GND);

UUT_POINT      : J22, LIMIT=(VOLT,  7.00000E+01,  0.00000E+00, GND);

UUT_POINT      : J19_B, ALIAS=GND;

  /*************************************************/
  /*                                               */
  /*  ATE FUNCTIONS                                */
  /*                                               */
  /*************************************************/


FUNCTION    20: CONST_R, FUNCTION TYPE=M, #PINS= 2,
        PARAM_01=(X, T, LIMIT=(OHM,  1.00000E+03,  1.00000E+00)),
        VALUE RETURNED='TRUE/FALSE';

FUNCTION   120: AMPL_TOL, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    40: DISTORTION, FUNCTION TYPE=M, #PINS= 2,
        PARAM_01=(X, T, LIMIT=(%,  1.00000E+75, -1.00000E+75)),
        PARAM_02=(Y, S, LIMIT=(KHZ,  1.00000E+02,  0.00000E+00)),
        VALUE RETURNED='TRUE/FALSE';

FUNCTION   140: DISTORT, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    50: SIGNAL_AM, ALIAS=SAM, FUNCTION TYPE=S, #PINS= 1,
        PARAM_01=(X, S, LIMIT=(MHZ,  1.00000E+02,  1.00000E-01)),
        PARAM_02=(Y, S, LIMIT=(DB, -1.00000E+01, -1.50000E+02)),
        PARAM_03=(Z, S, LIMIT=(°,  1.00000E+75, -1.00000E+75)),
        PARAM_04=(W, S, LIMIT=(KHZ,  1.50000E+01,  1.00000E-01));


FUNCTION    30: SINE_D, ALIAS=SINE_DELAY, FUNCTION TYPE=M, #PINS= 2,
        PARAM_01=(X, T, LIMIT=(VOLT,  1.00000E+01, -1.00000E+01)),
        PARAM_02=(Y, T, LIMIT=(MHZ,  1.00000E+01,  0.00000E+00)),
        PARAM_03=(Z, S, LIMIT=(SEC,  1.00000E+75, -1.00000E+75)),
        COMMENTS='APMPL., FREQ., TIME DELYD';

FUNCTION   130: REF_VOLT, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    10: CONST_S, FUNCTION TYPE=S, #PINS= 2,
        PARAM_01=(X, S, LIMIT=(VOLT,  6.00000E+01,  0.00000E+00)),
        VALUE RETURNED=' CONSTANT VOLT.';

FUNCTION   110: FREQ_TOL, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

  /*************************************************/
  /*                                               */
  /*  ATE CONNECTION POINTS                        */
  /*                                               */
  /*************************************************/


ATE_POINT      : ATE_J24B, UUT_POINTS=(J24_B);

END MINIRADIOSET;
```

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- DIAGNOSES <=> TEST-MODULES

DIAGNOSIS	TEST-MODULES
02	DC_INPUT
03	DC_INPUT
07	AMPL
08	AMPL
27	DISTORT_2W, DISTORT_10MW
30	DISTORT_2W
24	DISTORT_VOLT
25	DISTORT_VOLT
26	DISTORT_VOLT
04	FREQ
05	FREQ
06	FREQ
28	DISTORT_10MW

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- MESSAGES <=> DIAGNOSES <=> TESTS

MESSAGE	DIAGNOSES	TEST-MODULES
#%	D2	DC_INPUT
S/DESLPAY	D3, D7, 25, D6, 27	DC_INPUT, AMPL, DISTORT_VOLT, FREQ, DISTORT_2W, DISTORT_10MW
#&	D8, D5	AMPL, FREQ
#14	30, 28	DISTORT_2W, DISTORT_10MW
#15	24	DISTORT_VOLT
#17	26	DISTORT_VOLT
#5	D4	FREQ

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- AFFECTED-COMPONENTS <=> DIAGNOSES <=> TESTS

AFFECTED-COMPONENT	DIAGNOSES	TEST-MODULES
1: INPUT_SHORT	D2	DC_INPUT
2: FREQ_TOL(STD_5MHZ_FRE)	D5	FREQ
3: AMPL_TOL(STD_5MHZ_FRE)	D8	AMPL
6: REF_VOLT(AUDIO_10MW)	26	DISTORT_VOLT
7: DISTORT(AUDIO_10MW)	28	DISTORT_10MW
00603: DISTORT(AUDIO_2W)	30	DISTORT_2W

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- UUT-CONNECTING-POINTS <=> TEST-MODULES <=> ATE-CONNECTING-POINTS

UUT-CONNECTING-POINT	TEST-MODULES(S/M)	ATE-CONNECTING-POINTS
J24_B/XJ24_B	DC_INPUT(M), FREQ(S), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	ATE_J24B
J24_C/GND	DC_INPUT(M), DISTORT_2W(M), DISTORT_VOLT(M), FREQ(S), FREQ(M), DISTORT_10MW(M), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	
J19_L	DISTORT_2W(M)	
J16	DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	
J19_A	DISTORT_VOLT(M), DISTORT_10MW(M)	
J22	FREQ(M)	
J19_B/GND	DISTORT_2W(M), DISTORT_VOLT(M), FREQ(S), FREQ(M), DISTORT_10MW(M), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)	

SUMMARY CROSS-REFERENCES, FILE: XREF2 --- ATE-FUNCTIONS <=> TEST-MODULES

ATE-FUNCTION,TYPE	TEST-MODULES(S/M)
CONST_R, M	DC_INPUT(M)
DISTORTION, M	DISTORT_2W(M), DISTORT_10MW(M)
SIGNAL_AM/SAM, S	DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)
SINE_D/SINE_DELAY, M	DISTORT_VOLT(M), FREQ(M)
CONST_S, S	FREQ(S), DISTORT_VOLT(S), DISTORT_2W(S), DISTORT_10MW(S)

INTRA MODULE SEQUENCING DC_INPUT
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5
1	$M_W0001	CONJUNCTION	0	0	0	0	1
2	$M_W0002	ASSERTION	0	0	0	0	0
3	D2	DIAGNOSES	0	0	0	0	0
4	D3	DIAGNOSES	0	0	0	0	0
5	MRES	VARIABLE	0	1	0	1	0

SEQUENCE OF PROCESSING FOR TEST DC_INPUT

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	1	0	$M_W0001	CONJUNCTION	(<J24_B, J24_C> = CONST_R(MRES OHM)) TARGET: MRES;
2	3	0	D2	DIAGNOSES	AFFECTED COMPONENTS = INPUT_SHORT, TYPE = #4;
3	5	1	MRES	VARIABLE	LOCAL
4	2	2	$M_W0002	ASSERTION	MRES > 100 SOURCE: MRES;
5	4	2	D3	DIAGNOSES	OTHER PARAMETERS=(' OHMS', MRES), TYPE = D;

INTRA MODULE SEQUENCING AMPL
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4
1	$M_W0001	ASSERTION	0	0	0	0
2	D7	DIAGNOSES	0	0	0	0
3	D8	DIAGNOSES	0	0	0	0
4	V1	VARIABLE	1	1	0	0

SEQUENCE OF PROCESSING FOR TEST AMPL

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	3	0	D8	DIAGNOSES	AFFECTED COMPONENTS = AMPL_TOLISTD_5MHZ_FRE), OTHER PARAMETERS=('AMPL'), TYPE = #6;
2	4	0	V1	VARIABLE	GLOBAL / SOURCE /
3	1	1	$M_W0001	ASSERTION	V1 = 0.26 +- 0.06 SOURCE: V1;
4	2	1	D7	DIAGNOSES	OTHER PARAMETERS=('VRMS', V1), TYPE = D;

INTRA MODULE SEQUENCING DISTORT_2W
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6
1	$S_W0001	CONJUNCTION	0	2	0	0	0	0
2	$M_W0001	CONJUNCTION	0	0	0	0	0	1
3	$M_W0002	ASSERTION	0	0	0	0	0	0
4	27	DIAGNOSES	0	0	0	0	0	0
5	30	DIAGNOSES	0	0	0	0	0	0
6	P1	VARIABLE	0	0	1	1	0	0

SEQUENCE OF PROCESSING FOR TEST DISTORT_2W

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	1	0	$S_W0001	CONJUNCTION	(<J24_B, GND> = CONST_S(27.5 VOLT)) &(<J16> = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ));
2	5	0	30	DIAGNOSES	AFFECTED COMPONENTS = DISTORT(AUDIO_2W), OTHER PARAMETERS=(' 2W', 5.0), TYPE = #18;
3	2	1	$M_W0001	CONJUNCTION	(<J19_L, GND> = DISTORTION(P1 % ,1 KHZ)) TARGET: P1;
4	6	2	P1	VARIABLE	LOCAL
5	3	3	$M_W0002	ASSERTION	P1 <= 5 SOURCE: P1;
6	4	3	27	DIAGNOSES	OTHER PARAMETERS=('%', P1), TYPE = D;

INTRA MODULE SEQUENCING DISTORT_VOLT
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7	8
1	$S_W0001	CONJUNCTION	0	2	0	0	0	0	0	0
2	$M_W0001	CONJUNCTION	0	0	0	0	0	0	0	1
3	A1	ASSERTION	0	0	0	0	0	0	0	0
4	A2	ASSERTION	0	0	0	0	0	0	0	0
5	24	DIAGNOSES	0	0	0	0	0	0	0	0
6	25	DIAGNOSES	0	0	0	0	0	0	0	0
7	26	DIAGNOSES	0	0	0	0	0	0	0	0
8	V2	VARIABLE	0	0	1	1	0	1	0	0

SEQUENCE OF PROCESSING FOR TEST DISTORT_VOLT

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	1	0	$S_W0001	CONJUNCTION	(<J24_8, GND> = CONST_S(27.5 VOLT)) &(<J16> = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ));_
2	5	0	24	DIAGNOSES	TYPE = #15, TIME= 0.00000E+00, RESPONSE = ?;
3	7	0	26	DIAGNOSES	AFFECTED COMPONENTS = REF_VOLT(AUDIO_10MW), TYPE = #17;
4	2	1	$M_W0001	CONJUNCTION	(<J19_A, GND> = SINE_D(V2 VOLT ,*,0 SEC)) TARGET: V2;
5	8	2	V2	VARIABLE	LOCAL
6	3	3	A1	ASSERTION	V2 >= 2.2 SOURCE: V2;
7	4	3	A2	ASSERTION	V2 <= 2.8 SOURCE: V2;
8	6	3	25	DIAGNOSES	OTHER PARAMETERS=(' VAC', V2), TYPE = D;

INTRA MODULE SEQUENCING FREQ
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6	7	8	9
1	$S_W0001	CONJUNCTION	0	2	0	0	0	0	0	0	0
2	$M_W0001	CONJUNCTION	0	0	0	0	0	0	0	1	1
3	$M_W0002	ASSERTION	0	0	0	0	0	0	0	0	0
4	D4	DIAGNOSES	0	0	0	0	0	0	1	0	0
5	D5	DIAGNOSES	0	0	0	0	0	0	0	0	0
6	D6	DIAGNOSES	0	0	0	0	0	0	0	0	0
7	VAR1	VARIABLE	0	1	1	0	0	0	0	0	0
8	F1	VARIABLE	0	0	1	0	0	1	0	0	0
9	V1	VARIABLE	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR TEST FREQ

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	1	0	$S_W0001	CONJUNCTION	(<J24_B, GND> = CONST_S(27.5 VOLT));;
2	4	0	D4	DIAGNOSES	TYPE = #5, TIME= 0.00000E+00, RESPONSE=(VAR1);;
3	5	0	D5	DIAGNOSES	AFFECTED COMPONENTS = FREQ_TOL(STD_5MHZ_FRE), OTHER PARAMETERS=('FREQ'), TYPE = #6;;
4	7	1	VAR1	VARIABLE	LOCAL
5	2	2	$M_W0001	CONJUNCTION	(<J22, GND> = SINE_DIV(VOLT ,F1 HZ ,VAR1 SEC)) TARGET: F1, V1 SOURCE: VAR1;;
6	8	3	F1	VARIABLE	LOCAL
7	9	3	V1	VARIABLE	GLOBAL / TARGET /
8	3	4	$M_W0002	ASSERTION	IF VAR1=60 THEN F1 = 5E+06 +- 60 ELSE F1 = 5E+06 +- 2.5 SOURCE: VAR1, F1;;
9	6	4	D6	DIAGNOSES	OTHER PARAMETERS=(' HZ', F1), TYPE = D1

INTRA MODULE SEQUENCING DISTORT_10MW
ANALYSIS OF THE ADJACENCY MATRIX

			1	2	3	4	5	6
1	$S_W0001	CONJUNCTION	0	2	0	0	0	0
2	$M_W0001	CONJUNCTION	0	0	0	0	0	1
3	$M_W0002	ASSERTION	0	0	0	0	0	0
4	27	DIAGNOSES	0	0	0	0	0	0
5	28	DIAGNOSES	0	0	0	0	0	0
6	P1	VARIABLE	0	0	1	1	0	0

SEQUENCE OF PROCESSING FOR TEST DISTORT_10MW

ORDER VECT INDEX	ORDER VECTOR	RANK	NAME	TYPE	TEXT
1	1	0	$S_W0001	CONJUNCTION	(<J24_B, GND> = CONST_S(27.5 VOLT)) &(<J16> = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ));
2	5	0	28	DIAGNOSES	AFFECTED COMPONENTS = DISTORT(AUDIO_10MW), OTHER PARAMETERS=(' 10MW', 1.0), TYPE = #18;
3	2	1	$M_W0001	CONJUNCTION	(<J19_A, GND> = DISTORTION(P1 % ,2 KHZ)) TARGET: P1;
4	6	2	P1	VARIABLE	LOCAL
5	3	3	$M_W0002	ASSERTION	P1 <= 3 SOURCE: P1;
6	4	3	27	DIAGNOSES	OTHER PARAMETERS=('%', P1), TYPE = D;

RETURNED FROM SAP.
RETURNED FROM XREF1.
RETURNED FROM SOURCE2.
RETURNED FROM XREF2.
RETURNED FROM INTSEQ.
RETURNED FROM SEQANAL

END OF NOPAL PROCESSING.