UNIVERSITY of PENNSYLVANIA

PHILADELPHIA 19104

The Moore School of Electrical Engineering D2
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

Technical Report

NOPAL PROGRAM GENERATOR:

SYSTEM AND PROGRAMMING DOCUMENTATION

by

Project Staff

Automatic Program Generation Project
Department of Computer and Information Science
Moore School of Electrical Engineering
University of Pennsylvania
Philadelphia, Pennsylvania 19104

Prepared with Support From:
U.S. Army Communications and Electronics Command
ATE and Computer Software Support Branch
Code DRSEL-ME-SC
Fort Monmouth, New Jersey 07703
Under Contract DAAB07-81-F-1598

August 1982

Moore School Report

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
NOPAL PROGRAM GENERATOR: SYSTEM AND PROGRAMMING DOCUMENTATION	Technical Report
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Project Staff	DAAB07-81-F-1598
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer and Information Science Moore School of Electrical Engineering University of Pennsylvania Philadelphia, PA 19104	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS U.S. Army Communications & Electronics Material Readiness Command Fort Monmouth, NJ 07703	12. REPORT DATE August 1982 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) U.S. Army Communications & Electronics Command ATE & Computer Software Support Branch Code DRSEL-ME-SC Fort Monmouth, NJ 07703	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

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

Abstract Data Types	EQUATE ATLAS
ATLAS Test Programming Language	Fault Diagnosis
Automatic Test Program Generation (ATPG)	Modularity
Automatic Test Systems (ATS)	Non-Procedural Language
Directed Graphs	NOPAL

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

NOPAL is a descriptive language used to write specifications for testing a Unit Under Test (UUT) using an automatic test system (ATS). It can also be used for specification of general purpose computation tasks. The NOPAL system generates a program in the ATLAS test programming language that performs the specified testing.

A test specification in NOPAL consists of a number of modules, one of which is the main module. The main module contains the overall test specification. The other modules specify each an abstract data type consisting of a data

representation for the abstract data type and a set of operations, called modfuns, which can be performed on variables of the abstract data type. Interfaces among modules are provided by means of the abstract data types. Specification of modfuns is given non-procedurally by means of tests.

The main module in NOPAL consists of one or more tests. A test, in NOPAL, corresponds to the notion of a physical test on a UUT (Unit Under Test). It specified stimuli to be applied, measurements to be taken and logic for selecting a diagnosis depending on the passing or failing of the test. The specification of a test is given by conjunctions of stimuli and measurements and assertions of relations.

Information about the UUT and ATE (Automatic Test Equipment) can also be included in a module. This allows various interface checks to be performed.

The NOPAL program generator analyzes the specification of a module for consistency, completeness and non-ambiguity and generates a number of reports which serve as the documentation for the specification. Finally, if the specification is error-free, it generates a program in the EQUATE-ATLAS test programming language. Programs generated for various modules of a complete specification can simply be put together and executed on RCA EQUATE-ATLAS computer-controlled automatic test equipment.

The NOPAL program generator consists of three phases: (1) syntax analysis, (2) specification analysis, verification and sequencing, and (3) code generation. Each of the three phases are described in detail. An illustrative example is called MINI-RADIOSET is used throughout.

PREFACE AND ACKNOWLEDGEMENTS

The present report constitutes an updating of previous reports, "NOPAL Program Generator: System and Programming Documentation", by C. Tinaztepe, R. Sangal, H. Che and N. Prywes (December 1978) and by R. Sangal (March 1980). The NOPAL Program Generator has been changed extensively since the preparation of the above reports. It now has the following new capabilities:

1. Data Type definition, propagation and checking.

2. Definition of array type (repeating) tests and their efficient scheduling.

3. Definition of stimuli and measurement functions.

4. Use of digital stimuli and measurements functions and digital operations.

The previously prepared reports have been updated to include the above capabilities.

The staff engaged in the revisions and additions has contributed to this report as well. They consisted of:

Maya Gokhale
Chi-Ming Chen
Yuan Shi
Kwang-Shi Shu
Noah Prywes.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES (continued)

LIST OF FIGURES (CONTINUED)

LIST OF FIGURES (continued)

LIST OF FIGURES (continued)

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ALGORITHMS

CHAPTER 1
INTRODUCTION

1.1 Objective of the NOPAL System

The objectives of the NOPAL system 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 performance of this task automatically would alleviate these problems.

In order to define the objectives more precisely it is necessary to review the testing processes. An Automatic Test System (ATS) can be considered as composed of two components: (1) hardware which consists of the 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 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. The Automatic Program Generation Project at the

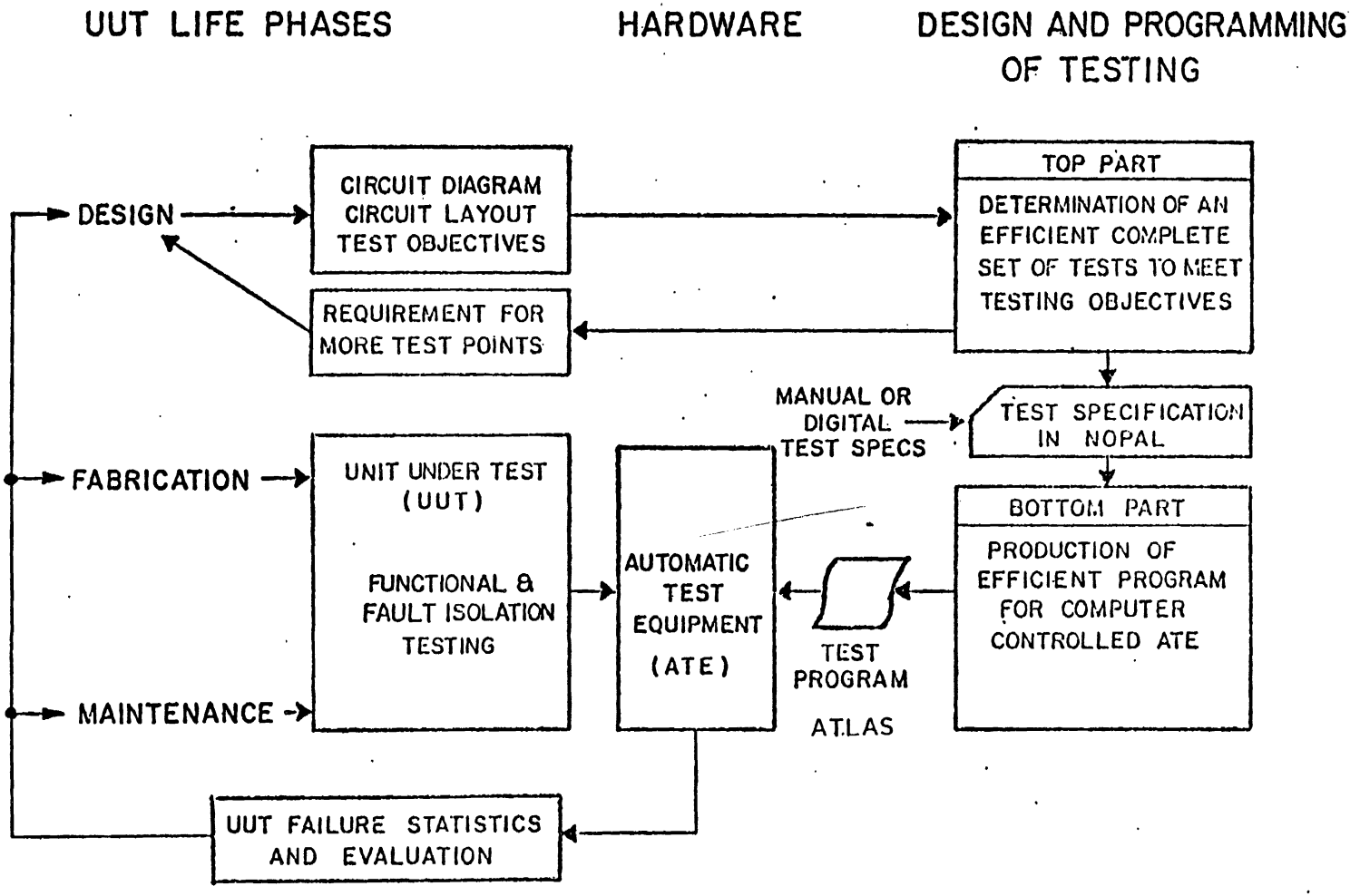


FIGURE 1.1 DESIGN AND USE OF TESTING IN THE LIFE CYCLE OF A UNIT UNDER TEST

University of Pennsylvania has undertaken the development of both parts. The top part is described in a report "Automatic Test Design," by C. Tinaztepe ECOM 75-0650-F-2, June 1978. The present report concerns the bottom part only. It is a revision of a previous report "Automatic Test Program Generation" by Y.K. Chang ECOM-75-0650-F-1, March 1978, and incorporates description of extensive enhancements to the system.

1.2 Overview of the NOPAL Language and Program Generator

NOPAL 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 at test time: (1) a set of tests, (2) the UUT, and (3) the ATE.

A complete NOPAL specification can be broken up into subparts called modules. Each module has: (1) module functions (modfun for short) which contain tests, (2) the UUT description, and (3) the ATE description. The modules can be specified and a program generated independent of each other. All the modules except the main module specify an abstract data type. A modfun specified by a module can be called from other modules, in which case the tests specified by it are performed.

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, in a modfun, 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.

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

First, the language is non-procedural. The user can save much labor as compared to the use of procedural languages because the execution order of events or the control logic need not be specified. Nor does the user 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 individual sections of a NOPAL specification independent each of others. That is the test section is independent of the units under test and the object automatic test equipments sections. For instance, only the specification of ATE functions (which are procedures defined in the object test language) should be properly modified if another set of ATE is to be used.

Fourthly, the modules may be processed separately to produce programs. These programs may simply be put together to represent the complete specification. This makes the system highly modular.

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 sub-sections: 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 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.

The intuition and often unorganized thinking of the human programmer 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.

1.3 Organization of the Report

Chapter 2 is a user manual for the NOPAL language. The formal syntax of NOPAL is also provided in an extended BNF notation.

Chapter 3 gives an overview of the NOPAL Program Generation. It summarizes the three major phases of: (1) syntax analysis, (2) specification analysis, design, and sequencing, and (3)

code generation.

Chapter 4 describes the first phase-syntax analysis of the NOPAL specification. Descriptions of the organization of the simulated associative memory (where the specifications are stored), the formal specification of the NOPAL language, and a meta-processor which processes the language specification to generate a syntax analysis program are also provided.

Chapter 5 covers the second phase - specification analysis and sequencing. It describes the methodology and the algorithms of analysing and sequencing of each test module taken individually and of all test modules taken collectively.

Chapter 6 presents the final code generation. During this phase the results of analysis and sequencing of the test specification are used to generate a complete test program in the object language, EQUATE ATLAS.

At the end, there are two appendices. The first appendix is a pictorial description of the data structures of the associative memory. The second appendix is the listing of a NOPAL test specification input which is followed by the corresponding outputs including the reports of analysis and the EQUATE-ATLAS program generated by the NOPAL Processor.

CHAPTER 2

THE NOPAL LANGUAGE

This chapter is omitted and replaced by a separate report, the "NOPAL Reference Manual, Bottom-Part", May 1982, by Noah S. Prywes.

CHAPTER 3
THE NOPAL PROGRAM GENERATOR

3.1 Overview

This chapter presents an overall description of the NOPAL Program Generator. As described in Chapter 2, the NOPAL language is a non-procedural specification language which is used to describe tests for various classes of units-under-test (UUT) to be tested in conjunction with computer-controlled automatic test equipment (ATE).

The NOPAL program generator is designed to automate the program design, coding, and debugging phases of program development based on test specifications written in the NOPAL language. A collection of NOPAL statements describing a functional module is referred to as a NOPAL module. A complete description, in NOPAL of the tests desired for a given UUT under a given "virtual" ATE i.e. a collection of NOPAL modules, is referred to as a NOPAL specification. A NOPAL module consists of data declaration specification, modfun-specification, UUT-specification, and ATE-specification. A module-function-specification is the core of a NOPAL module. It consists of tests each of which describes the stimuli to be applied, measurements to be taken, logic to select the disgnoses, and the corresponding diagnoses and messages. A UUT-specification identifies: 1) Potential component failures and 2) the connecting points of the UUT. An ATE-sepcification defines: 1) the ATE functions, such as stimuli, measurements, or special computational capabilities, and 2) the ATE interconnecting points. As shown in Figure 3.1, the generator takes a NOPAL specification as input, and the performs analyses (syntax,

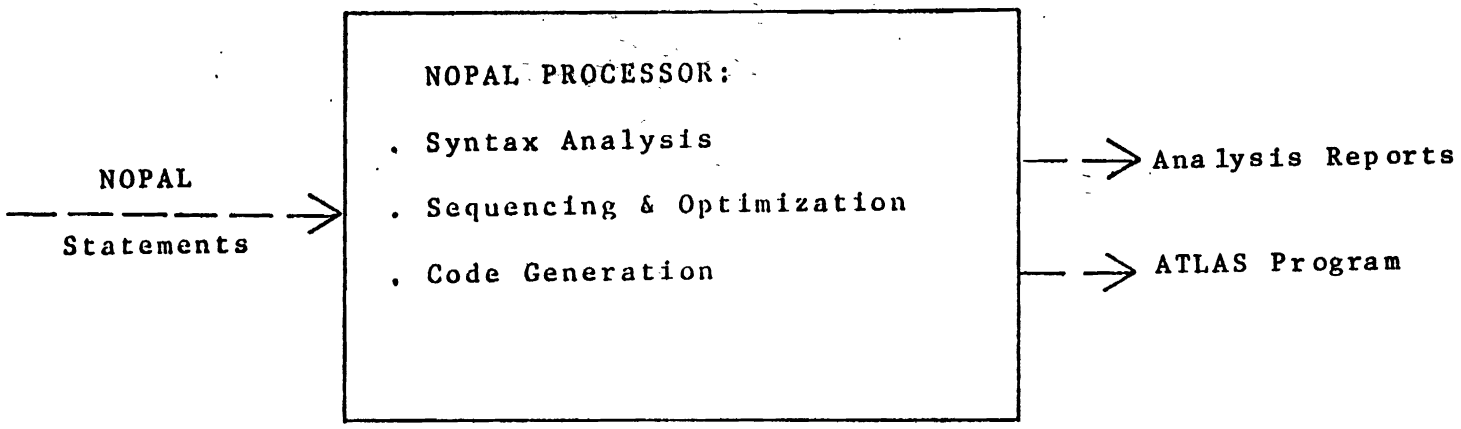


FIGURE 3.2: OVERVIEW OF NOPAL SYSTEM ~~PROCESSOR~~

semantics, completeness, consistency, ambiguity, etc.), test-modules sequencing (intra- and inter-test-module) and code generation. It finally produces, as output, a complete test program written in EQUATE-ATLAS. EQUATE-ATLAS is a subset of the IEEE Standard ATLAS Language. It also prints various user reports such as reformatted specification listing, cross-reference, sequencing flowchart, and error/warning messages. All of these output reports are described in the subsequent chapters. The printing of these reports can be inhibited by setting the appropriate parameter, when the Processor is invoked.

The Program generator performs the translation from source language (NOPAL) to target language (ATLAS). The program generator itself is implemented in PL/1. It processes a non-procedural source specification and generates a complete procedural target program, based on an application of directed-graph theory. It provides useful system-user interaction by sending to the user proper warnings/errors indicating necessary changes or additions to the user input.

The processing of the NOPAL specification consists of three major phases shown in Figure 3.2, which is a refinement of Figure 3.1. The three phases depicted in Figure 3.2 are briefly discussed in Sections 3.2, 3.3 and 3.4, and presented in greater detail in Chapters 4, 5 and 6 respectively.

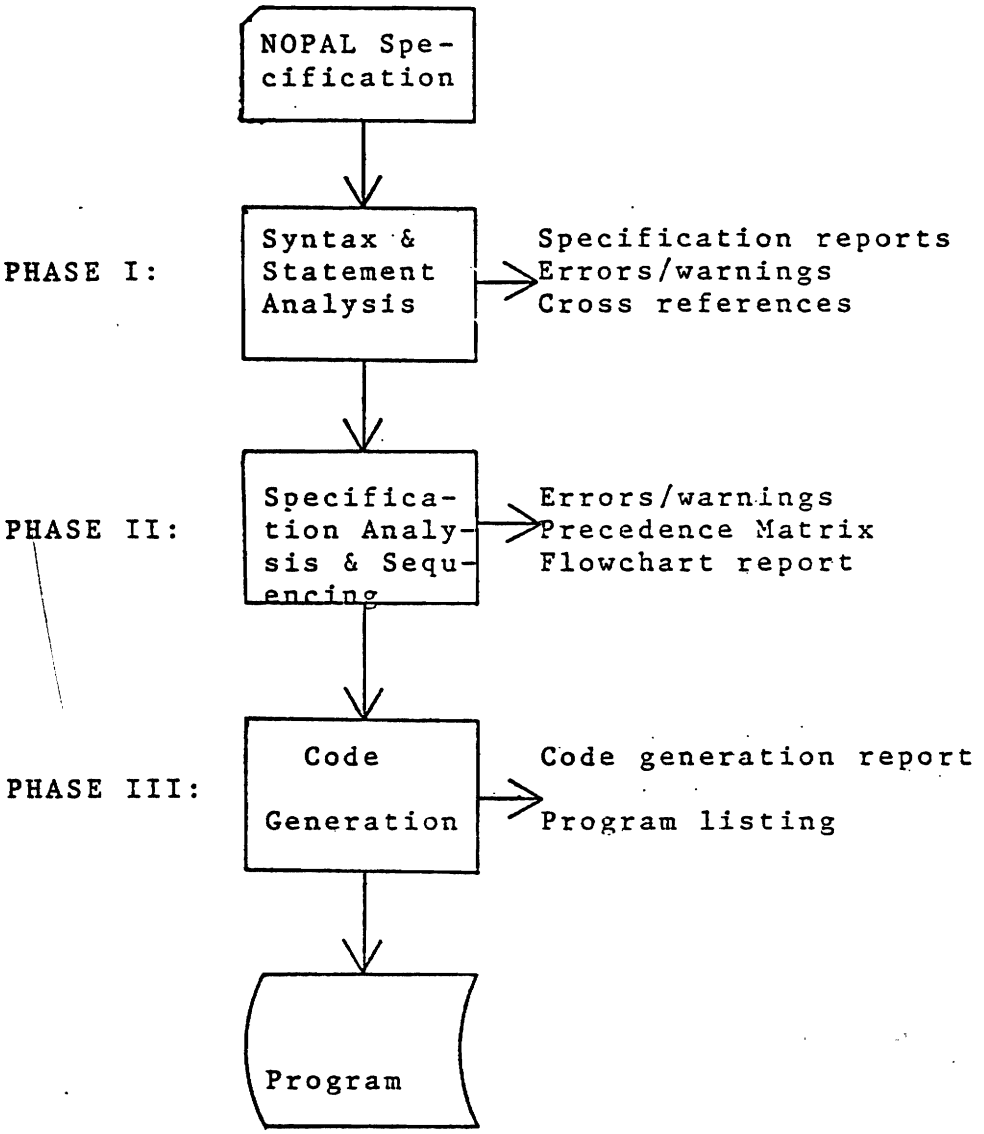


FIGURE 3.2: MAJOR PHASES OF NOPAL PROCESSOR

3.2 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 sub-monotor, called Syntax Analysis Program (SAP), which is itself generated automatically by a meta-processor called Syntax Analysis Program Generator (SAPG). The input to SAPG is the formal syntax rules of the NOPAL language with references to user-written routines. Changes to the syntax of NOPAL during development or in the future can thereby be conveniently made. Once SAP is generated, SAPG is no longer needed. SAP consists of routines such as lexical analyzer, error-stacker, and store/retrieve package.

In this phase, the consistency of the use of data types is checked. This is done by the derivation of data types in an "operator-based fashion". The results of derived data types are entered into a directory and an error message will be issued if any inconsistent use of data types is discovered.

Another task of this phase is to store the NOPAL statements in an associative memory (stimulated in main memory) for ease in subsequent retrieval and modification of the NOPAL specification during the later stages of analysis and processing. The associative memory is also used for generating three reports. A conventional cross-reference-and-attributes report is produced together with some warnings and error messages. A completely reorganized, easy to read source listing of the NOPAL specification is also printed. 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 4.

3.3 Phase II: Specification Analysis And Sequence Determination

This phase of the Processor determines the precedence relationships based on the 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 the precedence relationships. These relationships are used to form a directed graph, represented by a precedence matrix. Each node of the graph represents a data name (variable), a diagnosis, 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 or not the test specifications are complete, consistent, and/or unambiguous. Also, a user report

is produced containing the error/warning messages, the 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 and repetitions 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 flowchart-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 NOPAL specification, considering each test module as an integral unit.

Detailed description of this phase is covered in Chapter 5.

3.4 Phase III: Code Generation

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

Second, ATLAS procedures are defined for every test module and diagnosis specified by the user; the code for a test-module subroutine is generated based on the internal sequencing of the test module. Finally, the ATLAS code for properly invoking the test-module procedures and for the necessary control logic are generated. This step is performed based on the inter-test-module (i.e., external) sequencing of the specification.

The product of this phase is an ATLAS test program in accordance with the NOPAL specification provided by the user for testing the UUT.

The ATLAS program is further augmented by a library of ATLAS functions or procedures. Any routine (NOPAL function) which is used in the NOPAL specification and whose ATLAS code is not supplied at this stage by the user is expected to be included later at ATLAS compile time. The complete ATLAS program is then ready for compilation and execution (actual testing of the UUT) in a given ATE system. Chapter 6 describes Phase III in greater detail.

CHAPTER 4

SYNTAX ANALYSIS AND ASSOCIATIVE MEMORY

4.1 Overview

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 4.1, 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 4.2 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 Service Routines. These are covered in Section 4.3.

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 4.4.

Section 4.5 describes the processing of two NOPAL specifiation 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

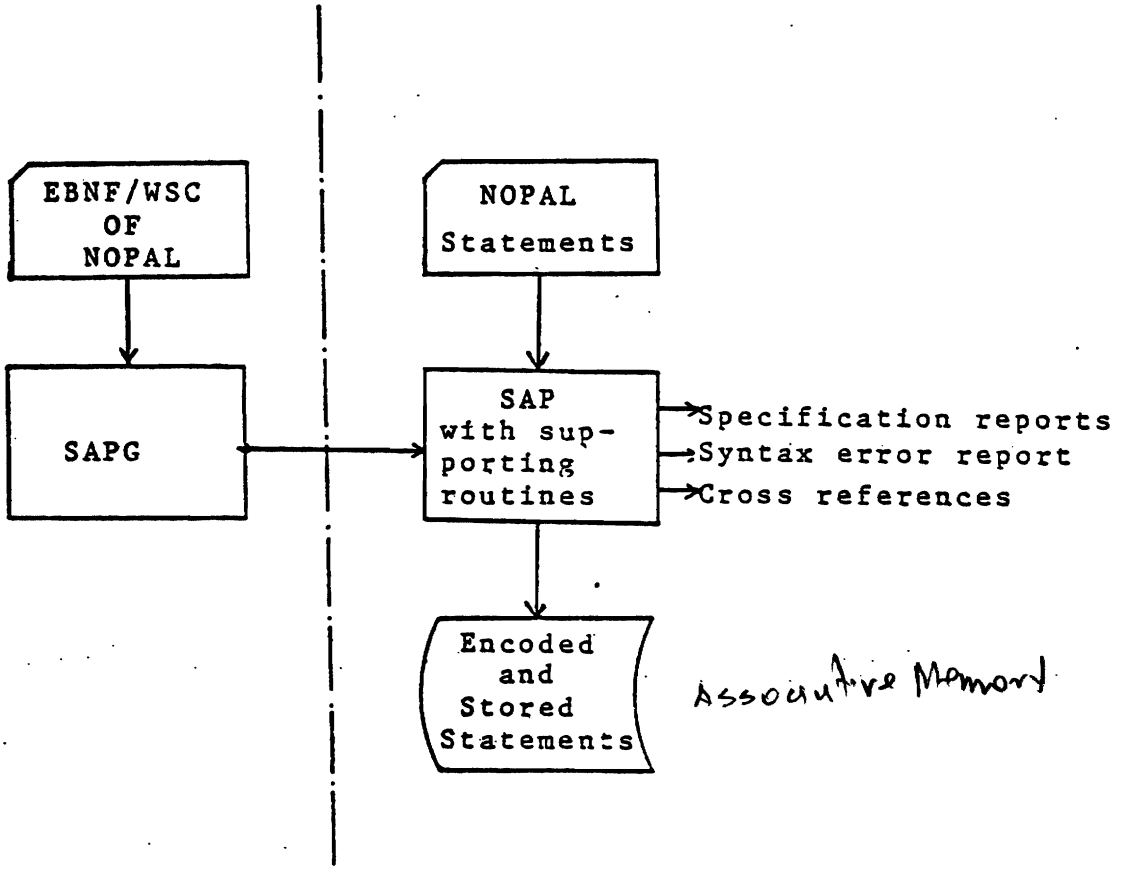


FIGURE 3.4 ~~4.1~~: SAP WITH SAPG AND OUTPUTS

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

Finally, Section 4.6 presents the part of the system that generates a number of cross reference reports, which are summarized in Table 4.1.

4.2 Syntax Analysis Program - SAP

The Syntax Analysis Program (SAP) for parsing 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. A brief review of SAPG and EBNF/WSC is given in the following sections.

4.2.1 Extended Backus Normal Form with Subroutine Calls (EBNF/WSC) Specification of NOPAL

The EBNF/WSC includes and extends the concepts of the conventional Backus Naur Form (BNF). The BNF itself is a language consisting of statements (production rules) which describe the syntax of formal languages (here NOPAL). BNF used four meta-linguistic characters <,>, ::=, and |. Sequences of characters enclosed in angle-brackets are called non-terminals, and denote names of the syntactic units and for which substitutions can be made. Sequences of characters not enclosed in angle-brackets are called terminals, which represent keywords or characters in NOPAL.

Report Number	Name of Report	Name of Entity Cross References	Name of Entities Cross Referenced with	Data Type
1	XREF-ATTR	Variable, data	Source statement numbers, attributes	Decimal
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	Boolean

TABLE 4.1. CROSS REFERENCE REPORTS.

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 "|".

To facilitate language description, BNF was extended to EBNF with two meta-linguisitic symbols: [ ] representing optionality and [ ]* representing repetition of zero or more times.

A description of the NOPAL language using EBNF, without subroutine calls, has been presented in Figure 2.1. 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 call 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 4.2. Unlike the human-oriented EBNF specification presented in Figure 2.1, the EBNF/WSC specification includes the following two modifications:

(1) the EBNF specification has been restructured to conform to restrictions explained in Section 4.2.3, 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 4.2, 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 2.1. 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 EBNF and EBNF/WSC statements, the following is an example from the EBNF specification of Figure 2.1.

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

In the EBNF/WSC specification on lines 115-116 of the Figure 4.2, 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.

4.2.2 Sap Generator - SAPG

As indicated in Figure 4.1, SAPG produces SAP based on the EBNF/WSC specification of the NOPAL language. The operation of SAPG is briefly summarized in the following paragraphs.

APPENDIX: EBNF/WSC SPECIFICATION OF THE NOPAL LANGUAGE


```
/* ---------------------- THE GLOBAL SYNTACTICAL STRUCTURE ---------------------- */

<NOPAL_SPECIFICATION> ::= [ <NOPAL_STMTS> /CLRERRF/ ]*
                          /STMT_FL/ <NOPAL_SPECIFICATION>
<NOPAL_STMTS>     ::= /E(34)/[NOPAL] <SPECIFICATION>[<SPEC_NAME>/SVLBL/]
                                          /STSPEC/ /STMTEND/
                    | <TEST_MODULE_SPEC>
                    | <DATA_DCL_SPEC>
                    | <UUT_SPEC>
                    | <ATE_SPEC>
                    | END [<SPEC_NAME>/SVLBL/] /STEND/ /STMTEND/
<SPECIFICATION>   ::= /SPECIF/
<SPEC_NAME>       ::= <LABEL>
<LABEL>           ::= /LABEL/


/* ---------------------- TEST MODULE SPECIFICATION ---------------------- */

<TEST_MODULE_SPEC>::=<TEST>[<TEST_LABEL>/SVLBL/]
                [ ( /CRTSUB/ <T_SUBEXPR> /E(5)/)/ADDRPAR/]
                /MFTEST/ /STTEST/ /STMTEND/
              | <STIMULI> <WAVEFORM_ID>/STSTIM/ [:<BACK_REF>] /STMTEND/
              | <MEASURE> <WAVEFORM_ID>/STMEAS/ [:<BACK_REF>] /STMTEND/
              | LOGIC     <WAVEFORM_ID> [:] <LOGIC_DIAG_LIST>/STLOG/ /STMTEND/
              | <WAVEFORMS> [<WAVEFORMS>]*
              | <DIAGNOSIS_DEFINITION> [<DIAGNOSIS_DEFINITION>]*
              | <MESSAGE_DEFINITION> [<MESSAGE_DEFINITION>]*

<TEST>          ::= TEST | MODFUN
<TEST_LABEL>    ::= <LABEL>
<T_SUBEXPR>         ::= <T_SUB> [ , <T_SUB> ]*
<T_SUB>             ::= /E(1)/ <ARITH_EXPR> /GTTSUB/


<STIMULI>       ::= /STIMULI/
<MEASURE>       ::= /MEASURE/


<WAVEFORM_ID>  ::= [<LABEL>/SVLBL/] [(/E(38)//E(5)/<LABEL>/SVLBL2/)]

<BACK_REF>      ::= /CONJ1/<BACK_REFERENCE> [<DECLARATION>]* /STWAVFM/
<BACK_REFERENCE>  ::= [SAME] AS/E(33)/ <LABEL> /BRSLBL/
                          [<EXCEPT> <SIMPLE_CONJUNCTION>]

<LOGIC_DIAG_LIST>::= /LOGID/<LOGOP_DIAGLBL> [,<LOGOP_DIAGLBL>]*
<LOGOP_DIAGLBL>   ::= /E(25)//E(26)/<LOGICAL_OPERATOR> <LABEL>/SLOPLBL/
                        [ <D_SUBEXPR> ]
<LOGICAL_OPERATOR> ::= /LOGICOP/
<D_SUBEXPR>         ::= [ ( <D_SUB> [ , <D_SUB> ]* /RPAR/)] /STDSUB/
<D_SUB>             ::= /E(1)/ <ARITH_EXPR> /GTDSUB/

<WAVEFORMS>     ::= <CONJUNCTION>
                  | <ASSERTION> [<ASSERTION>]*
<CONJUNCTION> ::= <CONJUNCT><WAVEFORM_ID> /E(2)/ : /CONJ1/ <CONJUNCTION_BODY>
                                  [<DECLARATION>]* /STWAVFM/ /STMTEND/
<CONJUNCT>      ::= /CONJUNC/
<CONJUNCTION_BODY>  ::= <BACK_REFERENCE> | <TRIPLET_CONJUNCT>
<TRIPLET_CONJUNCT>  ::= <IF_CONJUNCTION> | <SIMPLE_CONJUNCTION>
<RELATION>      ::= /RELREC/
```

Figure 4.2 EBNF/WSC

```
<IF_CONJUNCTION>   ::=  <IF_CLAUSE> <SIMPLE_CONJUNCTION>
                                [ELSE <TRIPLET_CONJUNCT>]

<SIMPLE_CONJUNCTION>::=  <TRIPLET> [& <TRIPLET>]* /STRIPT/
<TRIPLET>::= (<CONN_DIM_EX>/CONJ5/<RELATION>/CJSREL/<FUNC_DIM_EX>/CJSFDE/)
           |<CONN_DIM_EX>/CONJ5/<RELATION>/CJSREL/<FUNC_DIM_EX>/CJSFDE/
<CONN_DIM_EX>      ::= <CONNECTOR> [<DIMENSION>/CDESDM/]
<CONNECTOR>        ::= <LBRACKET> <CONNECTOR_ID>/CNXSUBS/
                       [,<CONNECTOR_ID>/CNXSUBS/]* /E(10)/ <RBRACKET>
                     | <CONNECTOR_ID> /CNXSUBS/
<LBRACKET>  ::= /LTA/
<RBRACKET>  ::= /RTA/
<CONNECTOR_ID>     ::=  /E(9)/<IDENTIFIER>/CDESID/ /STCNX/
                      [( <ARGMENT>/STARG/
                       [,<ARGMENT>/STARG/]* /E(5)/ ) ] /FNEND/
<ARGMENT>          ::= <STRING_CONST> /STCHAR/
                      |<ARITH_OR_DIGIT_EXP>/FREES/

<FUNC_DIM_EX>      ::= /SETFDE/<IDENTIFIER>/E(19)//E(20)//FDES1//STMSFN/
                       [(/E(19)//E(20)/<FUNC_ARG> /STARG/
                        [,/E(19)//E(20)//ADDCOMA/<FUNC_ARG> /STARG/]*
                        /E(5)/ ) /ADDRPAR//E(19)//E(20)/] /FNEND/
<FUNC_ARG>         ::=  * /E(19)//E(20)/ /STSTAR/
                      | <STRING_CONST> /E(19)//E(20)/ /STCHAR/
                      | /OPINIT/
                        [<RELATION> /E(19)//E(20)/ /STOPO/] <VAL_DIM_EX>
                        [<PLUS_MINUS>/E(19)//E(20)//STOP1/ <VAL_DIM_EX> /STOP2/]
                        /VDESEND/
<VAL_DIM_EX>       ::=  <ARITH_EXPR>/FDESAX/ [<DIMENSION>/FDESDM/ /STDIM/]

<DECLARATION>      ::=  <VARIABLE_TYPE> [:] <VARIABLE_LIST>
<VARIABLE_TYPE>    ::=  /SRC_TGT/
<VARIABLE_LIST>    ::=  (<VAR_ELEM> [,<VAR_ELEM>]* /RPAR/)
                      | <VAR_ELEM> [,<VAR_ELEM>]*
<VAR_ELEM>         ::=  /E(14)/<IDENTIFIER>/DCLSID/ <SUBSCRIPT_LIST>

<ASSERTION>        ::=  <ASSERT> <WAVEFORM_ID> /ASRT1/ [:] <ASSERTION_BODY>
                        [<DECLARATION>]* /STWAVFM/ /STMTEND/
<ASSERT>           ::=  /ASSERT/

<ASSERTION_BODY>   ::=  <IF_ASSERTION>
                      | <SIMPLE_ASSERTION>
<IF_ASSERTION>     ::= <IF_CLAUSE> <SIMPLE_ASSERTION> [ELSE <ASSERTION_BODY>]
<SIMPLE_ASSERTION>::=  /GETASRT/ <RELATIONAL_EXPR>
                        [<PLUS_MINUS><ARITH_EXPR> /ASRANGE/ [%/ASPC/] ]
<RELATIONAL_EXPR> ::=  <ARITH_OR_DIGIT_EXP>
                       /ASEXPS/ /E(3)/ <RELATION>/ASREL/
                       <ARITH_OR_DIGIT_EXP> /ASEXPS/
<PLUS_MINUS>       ::=  /PLUSMIN/
<ARITH_EXPR>  ::= /SETAREX/ /ARINIT/
                  <TERM> [<ADD_OP>/AREXS1/ /STOP1/<TERM>/STOP2/]*
<TERM>        ::=  <FACTOR> [<MULT_OP>/AREXS1/ /STOP1/<FACTOR>/STOP2/]*
<SIGN>        ::=  <ADD_OP>
<ADD_OP>      ::=  + | -
<MULT_OP>     ::=  * | /
<EXPONENTIATION>  ::=  /EXPONET/
<FACTOR>      ::=  [^/STNOT/] [<SIGN>/AREXS1/ /STOPA0/]<PRIMARY>/CKNOT//STOPA3/
                   [<EXPONENTIATION>/AREXS1//STOP1/<PRIMARY>/STOP2/]*
```

Figure 4.2 (continued)

```
<PRIMARY>     ::=   <UNSIGNED_NUMBER>/AREXS1/ /STFLOAT/
                  | <FUNCTION_CALL>/AREXSVF/
                  | (<ARITH_EXPR> /AREXSAX/ /E(5)/ )
                  | <ARITH_EXPR> /AREXSAX/

<FUNCTION_CALL>  ::= <IDENTIFIER>/SETVF//STFNID/[(/VFS0/<ARGUMENT>/STARG/
                     [,/VFS1/<ARGUMENT>/STARG/]*/RPAR/)/VFS1/] /FNEND/
<ARGUMENT>       ::=  /E(1)/<STRING_CONST>/VFS2/ /STCHAR/
                     | * /VFS1/ /STSTAR/
                     | <ARITH_OR_DIGIT_EXP>/VFSAX/

<IF_CLAUSE>       ::=  IF /SETBEXP/ <BOOLEAN_TERM> /IFCOND/
                          /E(23)/ THEN /RESETB/
<BOOLEAN_TERM>    ::=  <BOOLEAN_FACTOR>
                       [ <OR>/BEXPS1/ /STOP1/<BOOLEAN_FACTOR>/STOP2/]*
<OR>              ::= /OR_OP/
<BOOLEAN_FACTOR>  ::=  <BOOLEAN_PRIMARY> [&/BEXPS1/ /STOP1/
                       <BOOLEAN_PRIMARY> /STOP2/]*
<BOOLEAN_PRIMARY> ::= <ARITH_OR_DIGIT_EXP>/BEXPSAX/
                      [<RELATION>/BEXPS1/ /STOP1/
                       <ARITH_OR_DIGIT_EXP>/BEXPSBX/ /STOP2/]

<ARITH_OR_DIGIT_EXP>::=/SETAREX/ /ARINIT/
                       <A_OR_D_TERM>
                       [<A_OR_D_OP1>/AREXS1/ /STOP1/<A_OR_D_TERM>/STOP2/]*
<A_OR_D_OP1>    ::=  <ADD_OP>
                   | <DIGIT_OR>
<DIGIT_OR>      ::=  /DIGITOR/

<A_OR_D_TERM> ::= <A_OR_D_FACTOR>
                  [<A_OR_D_OP2>/AREXS1/ /STOP1/<A_OR_D_FACTOR>/STOP2/]*
<A_OR_D_OP2>     ::=  <MULT_OP> | <DIGITOPS>
<DIGITOPS>       ::= /DIGOPS/

<A_OR_D_FACTOR>::=  [^/STNOT/][<D_EXPONENT>][<SIGN>/AREXS1/ /STOPA0/]
              <A_OR_D_PRIMARY> /CKNOT//STOPA3/
              [<EXPONENTIATION>/AREXS1//STOP1/<A_OR_D_PRIMARY>/STOP2/]*
<D_EXPONENT>   ::=  /ROSH/
<A_OR_D_PRIMARY>::= <HEX>
                  | <OCT>
                  | <BIN>
                  | <UNSIGNED_NUMBER> /AREXS1/ /STFLOAT/
                  | <FUNCTION_CALL>/AREXSVF/
                  | /E(8)/ (/SETP/ <BOOLEAN_TERM> /AREXSAX/ /E(5)/ ) /RESETP/
<HEX>   ::= /HEX/
<OCT>   ::= /OCT/
<BIN>   ::= /BIN/

<DIAGNOSIS_DEFINITION>  ::=  <DIAGNOSIS>/DIAG1/ <LABEL>/SVLBL/ [:]
                                        <DIAG_BODY> /STDIAG/ /STMTEND/
<DIAGNOSIS>         ::=  /DIAGNOS/
<DIAG_BODY>      ::=  [OPERATOR /E(15)//E(16)/<MESSAGE>:] <DIAG_KEYWD>
                          [,<DIAG_KEYWD>]*
<DIAG_KEYWD>   ::=  [AFFECTED] <COMP_FAIL> /E(24)/= <AFFECTED_COMPONENTS>
                 | [OTHER] <PARAMETER> /E(24)/= <OTHER_PARAMETERS>/SPARM/
                 | PRINT/E(24)/=/E(27)/<LABEL>/STYP/
                 | TIME /E(24)/= <TIMING>
                 | <RESPONSE> /E(24)/= <OPERATOR_RESPONSE>
```

Figure 4.2 (continued)

```
<AFFECTED_COMPONENTS>  ::=  <COMPONENT_ELEM> [<AND_OR>/CMPSLOP/
                                                <COMPONENT_ELEM>]* /CMPRLOP/

<OTHER_PARAMETERS>::=(<MSG_ARG_REQD> [,<MSG_ARG_REQD>]* /RPAR/)
                    | <MSG_ARGUMENT>
<MSG_ARG_REQD>     ::=  /E(31)/ <MSG_ARGUMENT>
<MSG_ARGUMENT>     ::=  <STRING_CONST>/MASTR/
                      | <NUMBER>/MANUM/
                      | <IDENTIFIER>/MAVAR/ <SUBSCRIPT_LIST>

<TIMING>           ::=  <NUMBER>/TIME1/ [<TIME_DIMENSION>/TIME2/]
<TIME_DIMENSION>   ::=  /TIMEDM/

<RESPONSE>     ::=  /RESPONS/

<OPERATOR_RESPONSE> ::= <OPERATOR> /OPRPS1/
                        | (<OP_VAR_LIST>/RPAR/)  [?/OPRPS1/]
                        | <OP_VAR>  [?/OPRPS1/]
<OPERATOR>         ::= /OPSYM/
<OP_VAR_LIST>      ::=  <OP_VAR> [,<OP_VAR>]*
<OP_VAR>           ::=  /E(32)/<IDENTIFIER>/OPRPS2/ <SUBSCRIPT_LIST>

<MESSAGE_DEFINITION>  ::=  <MESSAGE>/E(27)/<LABEL>/SVLBL/ [:]
                           [ALIAS /E(28)//E(29)//E(30)/ = <SYNONYM>/SVSYN/,]
                           [TEXT/E(24)/=] <MESSAGE_TEXT>/STMSG/ /STMTEND/
<SYNONYM>   ::=  <IDENTIFIER>
<MESSAGE_TEXT>  ::=  /E(39)/<TEXT_ELEM> [ [,]<TEXT_ELEM>]*
<TEXT_ELEM>    ::=  <CHAR_STRING>/TXTCH/


/* ---------------------- DATA DECLEARATION SPECIFICATION ---------------------- */

<DATA_DCL_SPEC>  ::= DCL /STRTDAT/ <DATA_DCL> /STDATA/ /STMTEND/
<DATA_DCL>       ::= <VAR_DCL>
                   | /MRSTRT/ <REC_DCL> [, <REC_DCL>]* /MRFIN/
<VAR_DCL>        ::=  <IDENTIFIER> /MDCL/ /MSCOPE/ /MFPID/
                      [, /MDCL//MSCOPE/ <IDENTIFIER> /MFPID/ /MLISTIN/ ]*
                       /E(16)/: <MFATTRIB> /MLISTCO/
<REC_DCL>        ::= <UNSIGNED_INTEGER> /MRDCL/ /E(37)/ <IDENTIFIER>/MFPID/
                       :<MFATTRIB>/MFIELD/
<MFATTRIB>       ::= /E(18)/ <DATA_TYPE> /MFDTYP/
                  [ ( <UNSIGNED_INTEGER> /MFDINT/ /E(5)/ ) ]
                  [ ARRAY /MFSTART/ ( <DIMN> /MFCHAR/[,<DIMN> /MFCHAR/]* /E(5)/ )
                    /RPAR/ /MFENAR/ ]
<DATA_TYPE>    ::= DEC | DECIMAL |
                   DIG | DIGITAL |
                   BOL | BOOLEAN |
                   INT | INTEGER |
                   <IDENTIFIER>
<DIMN>         ::= <LABEL> | *


/* ------------------------- UUT SPECIFICATION ------------------------- */

<UUT_SPEC>     ::=  <UUT_COMPONENT_FAILURE> [<UUT_COMPONENT_FAILURE>]*
                  | <UUT_CONNECTION_POINT> [<UUT_CONNECTION_POINT>]*
<UUT_COMPONENT_FAILURE> ::= <COMP_FAIL>[<COMP_FAIL_SEQN>/SVLBL/][:]/E(37)/
                    <COMPONENT>/UUTCMPF/[,<COMP_FAIL_KEYWD>]*/STCOMP//STMTEND/
```

Figure 4.2 (continued)

```
<COMP_FAIL_KEYWD> ::=  /E(35)/ /E(24)/ /E(36)/ ALIAS = <SYNONYM>/SVSYN/
                     | <FAILURE>[<FUNCTION>] = <IDENTIFIER>/FLSFF/
                     | <PARAMETER> = <PARM_LIST>/SVPARML/
                     | INDEX = <FAILURE_INDEX>/FLSIDX/
                     | <PROTECT> = <PROTECTION>
                     | <COMMENTS>
<PARM_LIST>       ::=  (<PARM_NAME> [,<PARM_NAME>]* /RPAR/)
                     | <PARM_NAME>


<PARM_NAME>       ::=  /E(37)/<IDENTIFIER> /PMSID/
<PROTECT>         ::=  /PROTECT/
<PROTECTION>      ::= (<COMP_LABEL> [,<COMP_LABEL>]* /RPAR/)
                     | <COMP_LABEL>
<COMP_LABEL>      ::=  <COMPONENT_ELEM>


<UUT_CONNECTION_POINT>  ::=  <UUT_POINT>[<UNSIGNED_INTEGER>/SVSEQ_/][:]/E(37)/
                        <IDENTIFIER>/UUTPNT/ /STRTSUB/[<CNXSUBSCRIPT>]/UUTSUBS/
                        [,<UUT_POINT_KEYWD>]*/STUUTPT//STMTEND/
<UUT_POINT>         ::=  /UUT_PNT/
<CNXSUBSCRIPT>::= ( /SUBSTRT/  <UNSIGNED_INTEGER>/STINT/
                          [,<UNSIGNED_INTEGER>/STINT/]* )   /SUBSEND/
<UUT_POINT_KEYWD>  ::=  /E(35)//E(24)//E(36)/ALIAS = <SYNONYM>/SVSYN/
                      | <CONNECT> = <UUT_CONNECTOR>
                      | LIMIT = <PROTECTIVE_LIMITS>
                      | <COMMENTS>
<CONNECT>           ::=  /CONNECT/
<UUT_CONNECTOR>     ::=  (/E(37)/<CONN_TYPE>/SVCNTYP/
                           [,/E(37)/<CONN_POINT>/SVCNPT/] /RPAR/)
                       | <CONN_TYPE> /SVCNTYP/
<CONN_TYPE>         ::=  <IDENTIFIER>
<CONN_POINT>        ::=  <IDENTIFIER>
<PROTECTIVE_LIMITS>::=  /GETLMT/([<DIMENSION>/PMSDM/] [,[<MAX_LIMIT>
                                /PMSHL/] [,[<MIN_LIMIT>/PMSLL/]
                            [,/E(37)/<REFERENCE_POINT>/PMSRPT/] ] ]/RPAR/)
                      | <DIMENSION>/PMSDM/
<MAX_LIMIT>   ::=  <NUMBER>
<MIN_LIMIT>   ::=  <NUMBER>
<REFERENCE_POINT>  ::=  <IDENTIFIER>


/* ------------------------- ATE SPECIFICATION ------------------------------ */

<ATE_SPEC>   ::=  <ATE_FUNCTION> [<ATE_FUNCTION>]*
                | <ATE_CONNECTION_POINT> [<ATE_CONNECTION_POINT>]*

<ATE_FUNCTION>::=<FUNCTION>[<UNSIGNED_INTEGER>/SVSEQ_/][:]/E(37)/<IDENTIFIER>
                /ATEFUNC//SAVENA/[,<FUNCTION_KEYWD>]*/STFUNC//STMTEND/
<FUNCTION_KEYWD>  ::=  /E(35)//E(24)//E(36)/ALIAS = <SYNONYM>/SVSYN/
                     | [<FUNCTION>] TYPE = <FUNCTION_TYPE>/FNSTYP/
                     | #PINS = <UNSIGNED_INTEGER>/FN_PINS/
                     | <PARAMETER> = <PARM>
                     | VALUE [RETURNED] = <ATTRIBP>
                     | <COMMENTS>
                     | <REQUIRE>/SREQUI/
<FUNCTION_TYPE>  ::=  /E(21)/ S | M | F | E
```

Figure 4.2 (continued)

```
<PARM>          ::= (<PARM_NAME> [, [ <ATTRIB> ]
                                   [,[LIMIT/E(24)/=]<PROTECTIVE_LIMITS>] ] /RPAR/)
                  | <PARM_NAME>


<ATTRIB> ::= <PARM_TYPE>/PMSTYP/, /E(18)/ <DATA_TYPE>/STDTYP/
             [ (<UNSIGNED_INTEGER>/STDINT/)  ]
             [ ARRAY /STRTAR/ ( <INTEGER> /STARAY/
               [, <INTEGER> /STARAY/]*
               /E(5)/ ) /ENDAR/ ]


<PARM_TYPE>     ::=  S | T

<ATTRIBF>         ::= /E(18)/ <DATA_TYPE>/STFDTYP/
                      [ (<UNSIGNED_INTEGER>/STDFINT/)  ]
                      [ ARRAY /STRTFAR/ ( <INTEGER> /STFARAY/
                                          [, <INTEGER> /STFARAY/]*
                                                 /E(5)/ ) /ENDFAR/ ]


<REQUIRE> ::= REQ = /E(36)/ <PHRASE1> [,/SAVECO/<PHRASE2>]*
            | REQUIRE = /E(36)/ <PHRASE1> [,/SAVECO/<PHRASE2>]*
<PHRASE1> ::= <VERB>/SAVEID/ <PHRASE3>
<VERB>    ::= APPLY | MEASURE | MONITOR | VERIFY | DO
<PHRASE2> ::= [<REQTXT> /SAVEID/]* [<NUM_OR_PARA>]*
              [<REQTXT> /SAVEID/]* [<NUM_OR_PARA>]*[<DIM_OR_IDEN>/SAVEID/]
<PHRASE3> ::= (/SAVELP/<PHRASE2>)/SAVERP/ | <PHRASE2>

<NUM_OR_PARA> ::= <NUMBER>/SAVEID/ | (/SAVELP/[<IDENTIFIER>/SAVEID/])/SAVERP/
<DIM_OR_IDEN> ::= <DIMENSION> | <IDENTIFIER>

<ACTION_VERB>     ::= <IDENTIFIER>
<CNX_LIST>        ::= [<IDENTIFIER> [( )]* ]*

<ATE_CONNECTION_POINT>  ::=  <ATE_POINT>[<UNSIGNED_INTEGER>/SVSEQ_/] [:] /E(37)/
                    <ATE_POINT_ID>/SVLBL/ /STRTSUB/[<CNXSUBSCRIPT>] /ATESUBS/
                     [,<ATE_POINT_KEYWD>]*/STATEPT//STMTEND/
<ATE_POINT>      ::=  /ATE_PNT/
<ATE_POINT_ID>   ::=  <IDENTIFIER>
<ATE_POINT_KEYWD>  ::=  /E(35)//E(24)//E(36)/ALIAS = <SYNONYM>/SVSYN/
                        | <UUT_POINT> = <UUT_POINTS>
                        | <COMMENTS>
<UUT_POINTS> ::=  (<IDENTIFIER>/SVPTID/ /STRTSUB/[<CNXSUBSCRIPT>]/AUSUBS/
          [,<IDENTIFIER>/SVPTID/ /STRTSUB/[<CNXSUBSCRIPT>]/AUSUBS/]* /RPAR/)
        | /E(37)/<IDENTIFIER>/SVPTID/ /STRTSUB/[<CNXSUBSCRIPT>]/AUSUBS/
```

Figure 4.2 (continued)

```
/* ------------------------------ MISCELLANEOUS ------------------------------ */

<AND_OR>              ::= /ANDOROP/
<CHAR_STRING>         ::= /CHARSTR/
<COMMENT>             ::= /COMMENT/
<COMMENTS>            ::= [<COMMENT> /E(24)/=] <CHAR_STRING>/SCOMT/
<COMP_FAIL>           ::= /COMPFL/
<COMP_FAIL_SEQN>      ::= <UNSIGNED_INTEGER>
<COMPONENT>           ::= <IDENTIFIER>
<COMPONENT_ELEM>      ::= <IDENTIFIER>/CMPSID/ [(/CMPSFF/<COMPONENT>/CMPSID/
                             [<AND_OR>/CMPSLOP/<COMPONENT>/CMPSID/]* /FFRSET/)]
                        | <COMP_FAIL_SEQN>/SVCMPFL/
<DIMENSION>           ::= /DIMREC/
<EXCEPT>              ::= /EXCEPT/
<FAILURE>             ::= /FAILURE/
<FAILURE_INDEX>       ::= <INTEGER>
<FUNCTION>            ::= /FUNCION/
<IDENTIFIER>          ::= /NAMEREC/
<INTEGER>             ::= /INTEGER/
<MESSAGE>             ::= /MESSAGE/
<NUMBER>              ::= /NUMBER/
<PARAMETER>           ::= /PARAMET/
<REQTXT>              ::= /REQTXT/
<STRING_CONST>        ::= /STRREC/
<SUBSCRIPT_LIST>      ::= [(<SUBSCRIPT> [,<SUBSCRIPT>]* /RPAR/)] /DCLSVAR/
<SUBSCRIPT>           ::= /E(1)/<ARITH_EXPR> /DCLSUBS/
<UNSIGNED_INTEGER>    ::= /POSINTG/
<UNSIGNED_NUMBER>     ::= /POSNUMB/
```

Figure 4.2 (continued)

SAPG itself is a small three-pass 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 boolean value. The procedure returns as 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 service 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 115-118 and 65 of Figure 4.2):

```
< 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 service routines, which are internal to the mechanisms of SAPG. Normally, they do not concern the language definer. These routines are further discussed in Section 4.3.6. 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 4.3.3. 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_DEF-INITION > . 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 sub-routine call /SVLBL/. Now a left bracket ([), signal-ling the beginning of an optionality group, is encounter-ed, 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.

4.2.3. 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 5.

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. 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 left recursion. 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.

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 of EBNF for NOPAL in Figure 2.1:

```
< 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 back-tracking 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.

4.3 Supporting Subroutines

A flowchart showing SAPG and SAP with the types of supporting subroutines is shown in Figure ~~4.3~~ 3.5. The manually-written supporting routines are of the following six types:

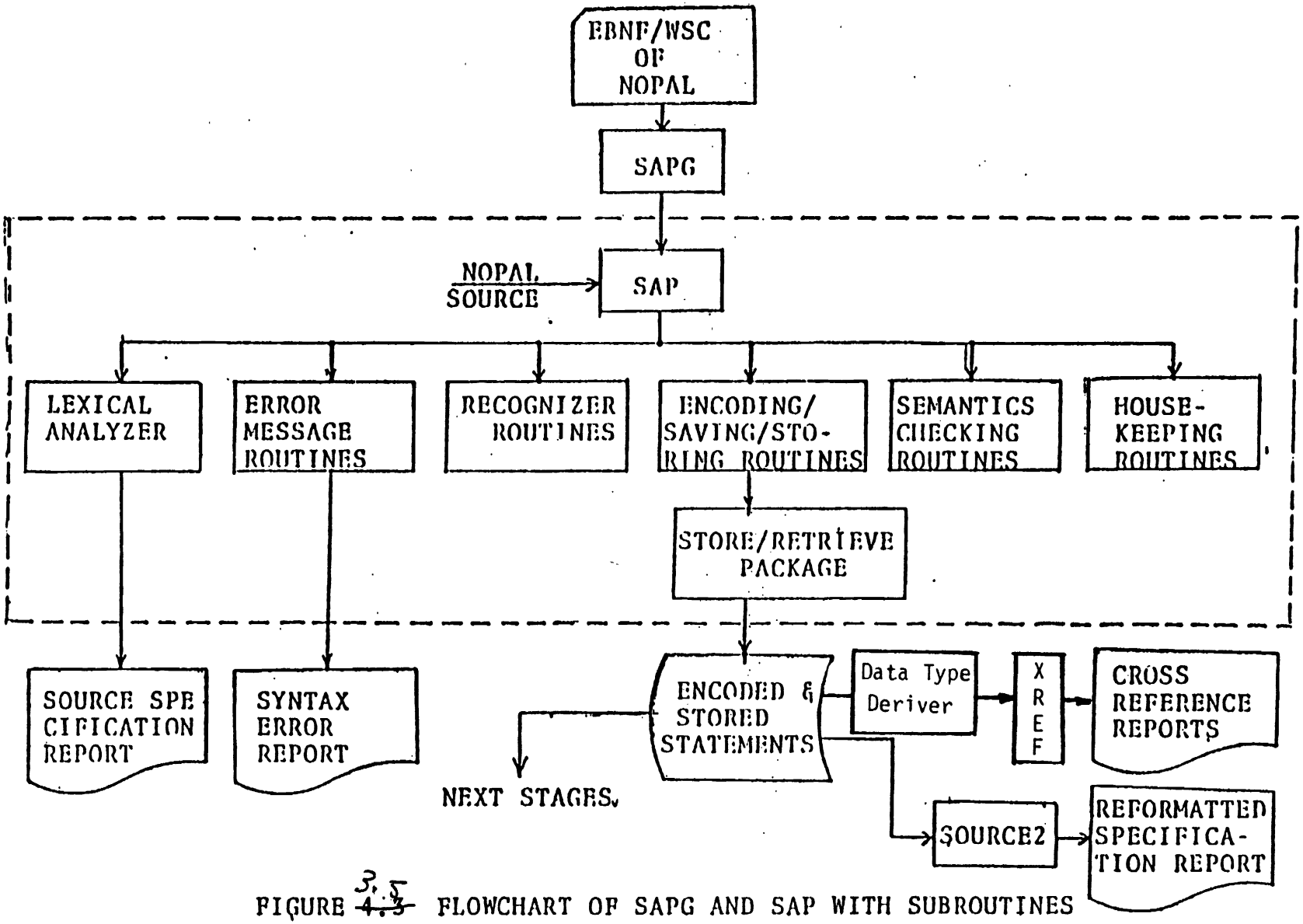


FIGURE ~~4.3~~ 3.5 FLOWCHART OF SAPG AND SAP WITH SUBROUTINES

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) service: required by the SAPG system in order to perform some internal 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 4.4.~~ ~~The~~ Two specification reports, source and reformatted, together with the syntax error report are ~~presented in Section 4.5,~~ ~~the~~ six cross reference reports ~~are indicated~~ are shown in Table ~~4.1~~ 3.1 ~~are discussed in Section 4.6.~~ insert Table 4.1 from p. 4-4

4.3.1 LEXICAL ANALYZER

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. end of insert

The lexical analyzer routine is based upon the concept of finite state machines. 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 4.2. The whole character set is divided into eleven categories such as letters, digits, delimiters, and special operators. The state transition diagram appears in Figure 4.4. Names enclosed by circles denote the states of the machine; two concentric circles

TABLE 4.2 CHARACTER CLASSES FOR NOPAL LANGUAGE

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

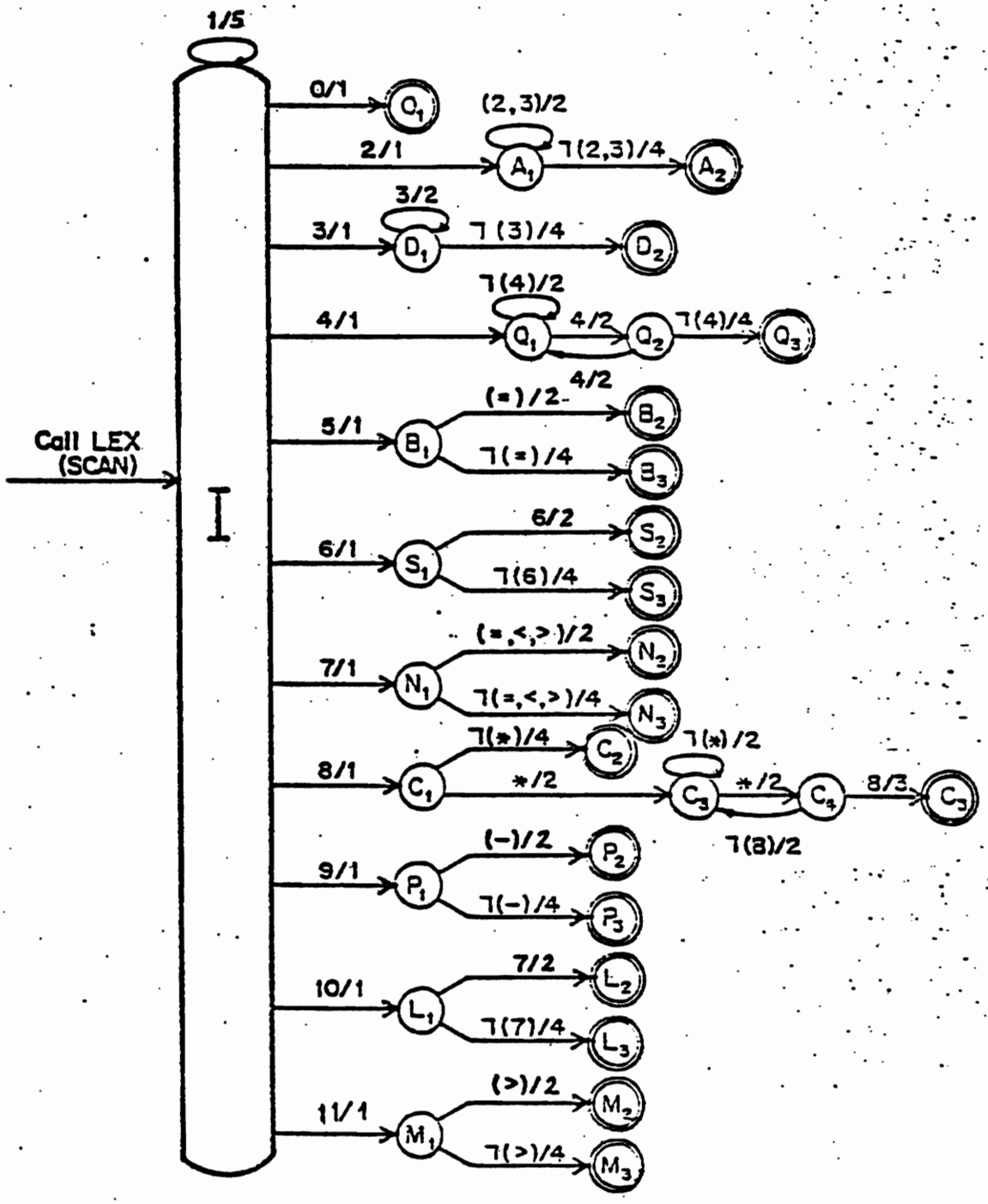


Figure 4.4 STATE TRANSITION DIAGRAM FOR LEXICAL ANALYZER

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 4.3). 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 4.2. 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 (a letter) or of class 3 (a digit), it takes action 2 (which concatenates 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 decrements the input pointer) and then goes to the next state. Table 4.3 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 4.3 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. 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 4.4, 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.

4.3.2 ERROR MESSAGE 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 interpretations are listed in Table 4.4.

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 4.4 ERROR STACKING ROUTINES, ERROR CODES AND MESSAGES

TABLE 4.4 (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
ERRDTY	DTYPER	missing data type
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 4.4 (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

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 4.3.3). To illustrate, consider the EBNF statement 102 of Figure 2.1:

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

The corresponding production rules in the EBNF/WSC of NOPAL (lines 153-158, 3, and 65 of Figure 4.2) 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.

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;
```

4.3.3 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 2.1) 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 NOAL are enumerated in Table 4.5.

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

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 4.5 RECOGNIZER ROUTINES

TABLE 4.5 (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
GIVE_RET	RETURN	In RETURN (.....
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, MAIN MODULE

TABLE 4.5 (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.

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 4.3.1) and '1'B (true) must be returned. Otherwise, '0'B (false) must be returned.

As indicated in Section 4.3.1, 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 4.3.1). 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.

4.3.4 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 4.4) 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 4.4. The encoding, saving and storing routines are summarized in Table 4.6 in the groups of statements.

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 4.6 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 4.6 (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.

The following store declaration for value returned:

STFDTYP	---	Stores the data type
STDFINT	---	Stores integer with data type e.g. char(10)array..
STRTFAR	---	Type in array
STFARAY	---	Stores the dimensions of array

TABLE 4.6 (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 4.6 (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 4.6 (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
	FDESMOD		Save modifier for func-dim-ex

Table 4.6 (continued)

ENDFAR --- End of array declaration

The following store parameter declaration:

STDINT --- Similar to STEDINT

STRRAR --- Similar to STRTFAR

STARAY --- Similar to STFARAY

ENDAR --- Similar to ENDFAR

STDTYP --- Similar to STDTYP

TABLE 4.6 (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 4.6 (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

16. The following procedures store the parse-tree of waveforms. Whenever the stack is used, stack (pointer type) PTREXPR is implied.

STOP1	---	Allocates a new node for the tree and enters in it the infix operator, which has been read in. Pops stack 'PTREXPR' and puts the pointer as first argument of the operator in the node. Finally pushes the pointer to the node on stack.

TABLE 4.6 (continued)

STOP2	---	Pops stack and enters the popped pointer in the second argument of the node pointed to by the top of stack, i.e. finishes building of the tree whose parts are on stack.
STFNID	---	Stores the function identifier which has just been encountered by allocating a new node. Pointer to node is pushed on stack.
STARG	---	Pops the stack and puts the pointer as the argument of the node of the new top of stack.
FNEND	---	FN call completed. Clean the node containing function and its arguments.
STCHAR	---	Stores char string in a new leaf node and pushes pointer to it on stack.
STFLOAT	---	Stores floating point number in a new leaf node and pushes pointer to it on stack.
STSTAR	---	Stores '*' in a new leaf node and pushes pointer on stack.
STCNX	---	Stores the connection point identifier in a new node and pushes pointer to it on stack.
STMSFN	---	Just like STFNID except that the function id is a stimuli or measurement fn.
ARINIT	---	Allocates the bit $SIGNSW which denotes whether optional <SIGN> has been found.

TABLE 4.6 (continued)

STOPAØ	---	Builds a node containing number 0, pushes it on stack and then calls STOP1.
STOPA3	---	If $SIGNSW is set then frees it and calls STOP2 otherwise just frees $SIGNSW.
STNOT	---	Stores the logical NOT operator and pushes pointer on stack.
STOPF4	---	Allocates the bit $FNMOD. This will denote whether the optional <FUNC-MODIFIER> is encountered.
STOPF5	---	Sets $FNMOD. Calls STOP1.
STOPF6	---	Frees $FNMOD. If it was set then calls STOP2.
OPINIT	---	Allocates $OPER. This will be set to true if optional <RELATION> operator is found in VDE i.e. <VAL-DIM-EXPR>.
STOPO	---	Sets $OPER. Enters the operator in the node and pushes on stack.
VDESEND	---	End of VDE reached. Free $OPER. If it was set then pop the stack and enter the popped pointer in the arg(1) of the node on the new top of stack.
STDIM	---	Creates node of type VDE. Pops the stack and puts the pointer as arg(1) of VDE. Arg(2) is set to point to the leaf containing dimension. Finally pushes the VDE node on stack.

TABLE 4.6 (continued)

17. The routines used for storing the subscripts of ATE & UUT points in their declaration are described below. The basic idea is the same as that described in (16) of Table 4.6 above. The stack used here is named PTRSUBS. It contains pointers which point to the list LIST_VAL.

This list contains the list of subscripts which occurred for ATE or UUT pts. Note that the subscripts must be integers.

SUBSTRT	---	Initializes the stack and other variables in preparation of connection point subscripts.
STINT	---	Stores integer in the list LIST_VAL at appropriate position (given by variable ARGSUBS).
SUBSEND	---	End of subscript found. Cleans LIST_VAL.
UUTSUBS	---	Enters the top of stack (which points to the list of subscripts) in structure of UUT_POINT.
ATESUBS	---	Same as above for ATE_POINT.
AUSUBS	---	Stores the subscripts of UUT's which occur in the declaration of ATE point in a temporary array.

Table 4.6 (continued)

18. For declarations of variables, structures and parameters in module function.

STRTDAT --- Data declaration begins - allocates storage DATASPEC.

STDATA --- Stores the entry DATASPEC associative memory.

MDCL --- Allocates storage for DCLDAT.

For data declaration of variables:

MSCOPE --- Marks scope of identifier as global.

MLISTIN -- Increments MVLIST which keeps count of this declaration (Not for structure declaration).

MLISTCO -- Enters same data type for all the variables found in declaration. (Not for structure declaration)

For declaration of structures:

MRDCL --- Allocates storage for DCLDAT. Prepare for record declaration.

MFIELD --- Checks whether it is a field.

MRSTRT --- Initializes stack for record declaration.

MRFIN --- Record over. Empty stack. Cleans the structures.

For attributes for the above and for parameters of module function.

MFDTYP --- Stores data type.

MFDINT --- Stores integer with data type e.g. char(5)---

MFSTART--- Prepares for array declaration for the data type.

MFCHAR --- Stores dimensions of the array

MFENAR --- Array ends. Cleans stricture.

For declarations of parameters of module function:

MFSRTG --- Stores type of parameter: source or target.

MFCID --- Stores identifier. For connection points as parameters.

Table 4-6 (continued)

For declarations of parameters of module function: (continued)

MFPID --- Stores identifier for parameters.

STMFP --- House keeping for parameters.

STMFC --- House keeping for connection points parameters.

STMFV --- House keeping for value return.

The routines used for storing the arithmetic and boolean expressions in a tree form are described below.

Each time an EBNF statement having expression, term, factor or primary, etc., on the left hand side of '::=' is invoked a tree corresponding to the input is built, and the top of the stack (called 'PTREXPR') points to the tree.

Consider the following example:

<TERM> ::=< FACTOR > [<MULT-OP>/AREXS1/ /STOP1/
<FACTOR> /STOP2/]*

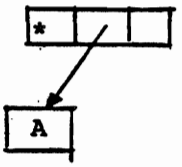
and input is A * (C+D). When the <TERM> is recognized, a tree corresponding to A*(C+D) is built and the pointer to the tree is placed on top of the stack. Let us follow this process token by token.

'A' is read in first. The rule for the non-terminal <FACTOR> is invoked and when it returns, there will be a tree having a single element 'A'. The pointer to the tree will be on top of the stack.

'*' is encountered next. The rule for <MULT-OP> is invoked which checks that the input '*' is indeed as expected. After this call is made to the procedure AREXS1. This does concatenation of the character string as described above in entry 15 of Table 4.6 and is not used for building the tree.

The next call is made to the procedure STOP1. This procedure knows that the first argument of the infix operator is on the top of stack and the infix operator ('*') has just been read in. So it produces a new node of the tree. The

infix operator is entered in the node (see the data structure 'EXPRS' for node of the tree). The stack is now popped and the popped pointer is entered as the first descendent of the node. At this stage we have a



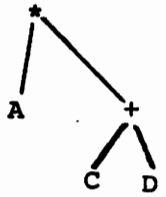
incomplete tree corresponding to the partial <TERM>. Pointer to this tree is pushed on stack.

<FACTOR> is invoked next, and on its completion root of the tree corresponding to (C+D) is pushed on stack.

Now call is made to procedure STOP2.

This procedure knows that the second argument is on top of stack and below it is the incomplete node. So it proceeds to pop the stack and enters the popped pointer as the second argument of the incomplete node, sitting now on the top of the stack.

Notice that <TERM> statement has been recognized and the pointer to the tree



is on the top of stack.

This same basic idea is utilized in

< ARITH_EXPR > , < FACTOR >, < PRIMARY > , < FUNCTION_CALL > ,

< ARGUMENT >,

< BOOLEAN_TERM >, < BOOLEAN_FACTOR > , < BOOLEAN_PRIMARY > ,

< CONNECTOR_ID >, < ARGMENT >> , < FUNC_TERM > , < FUNC_FACTOR > ,

< FUNC_PRIMARY > , < FUNC_ARG > , < CNXSUBSCRIPT > .

Some of the rules for above non-terminals have optional elements, and it is necessary to set flags when those optional elements are encountered.

List of the procedures named in EBNF/WSC together with a brief description is given as groups 16 and 17 in Table 4.6.

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 4.4. 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  SPECNAM CHAR(12) VAR INIT ('$SPEC00000');
```

```
    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.

4.3.5 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 $_{2.1}$ 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 4.7.

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.

The routines are subroutine-type PL/1 procedures. Inside such a routine, the lexical token and all the information encoded and/or saved previously by other routines may be accessed for the purpose of checking local semantics. To illustrate, again consider the

TABLE 4.7 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

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.

4.3.6 Service Routines

There are a few "service" 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. 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 service 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. The SAPG-generated SAP also uses five other service routines (each one is prefixed with "$"): $MARK, $POPF, "SUCCESS, $FAIL, and $PUSH. Except $PUSH, which is also used to push error codes in error-stacking routines (as indicated in Section 4.3.2), they are not useful to, and should not concer, the language definer. All service routines, with their functions, are summarized in Table 4.8.

4.4 Associative Memory And Store/Retrieve Sub-System

In order to facilitate the SAPG system with a generalized means for storing source statements and for later retrieval, the following subsystem has been implemented and included in the NOPAL Processor. The subsystem consists of two types of routines.

(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.

TABLE 4.8 SERVICE 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 4.8 SERVICE ROUTINES

TABLE 4.8 (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.

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 4.9. 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 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

TABLE 4.9 TYPES OF NAMES AND STATEMENTS

TYPE #	MNEMONIC	CLASS OF ENTITIES REPRESENTED
1	SPEC#	NOPAL specification label/statement
2	TEST#	Test module label/statement or modfun header.
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
17	DTYP#	Data-type name
18	REC#	Data declaration statement.

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.

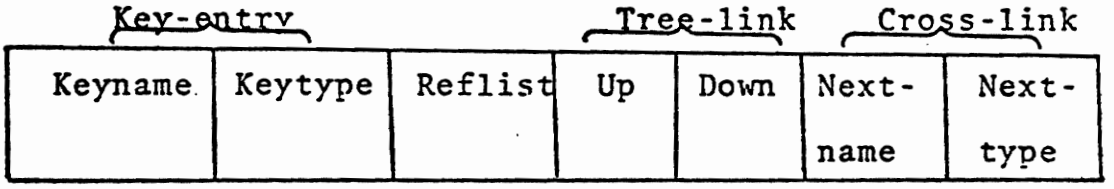
4.4.1 Directory And Storage Entries

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 4.5(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 denoting the type (or class)

(a) DIRECTORY ENTRY



(b) STORAGE ENTRY

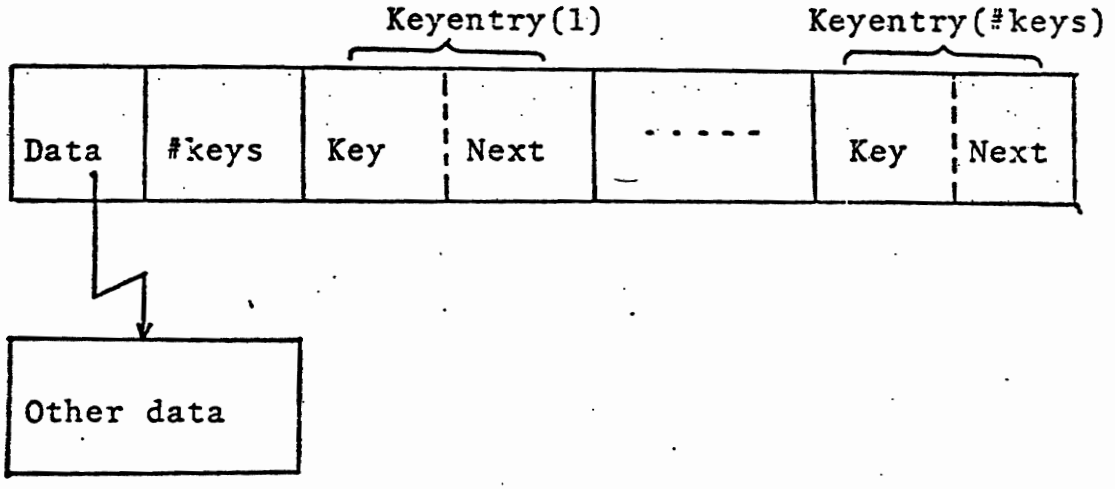


FIGURE 4.5 STRUCTURE OF THE DIRECTORY AND STORAGE ENTRIES

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 4.5(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 4.6 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

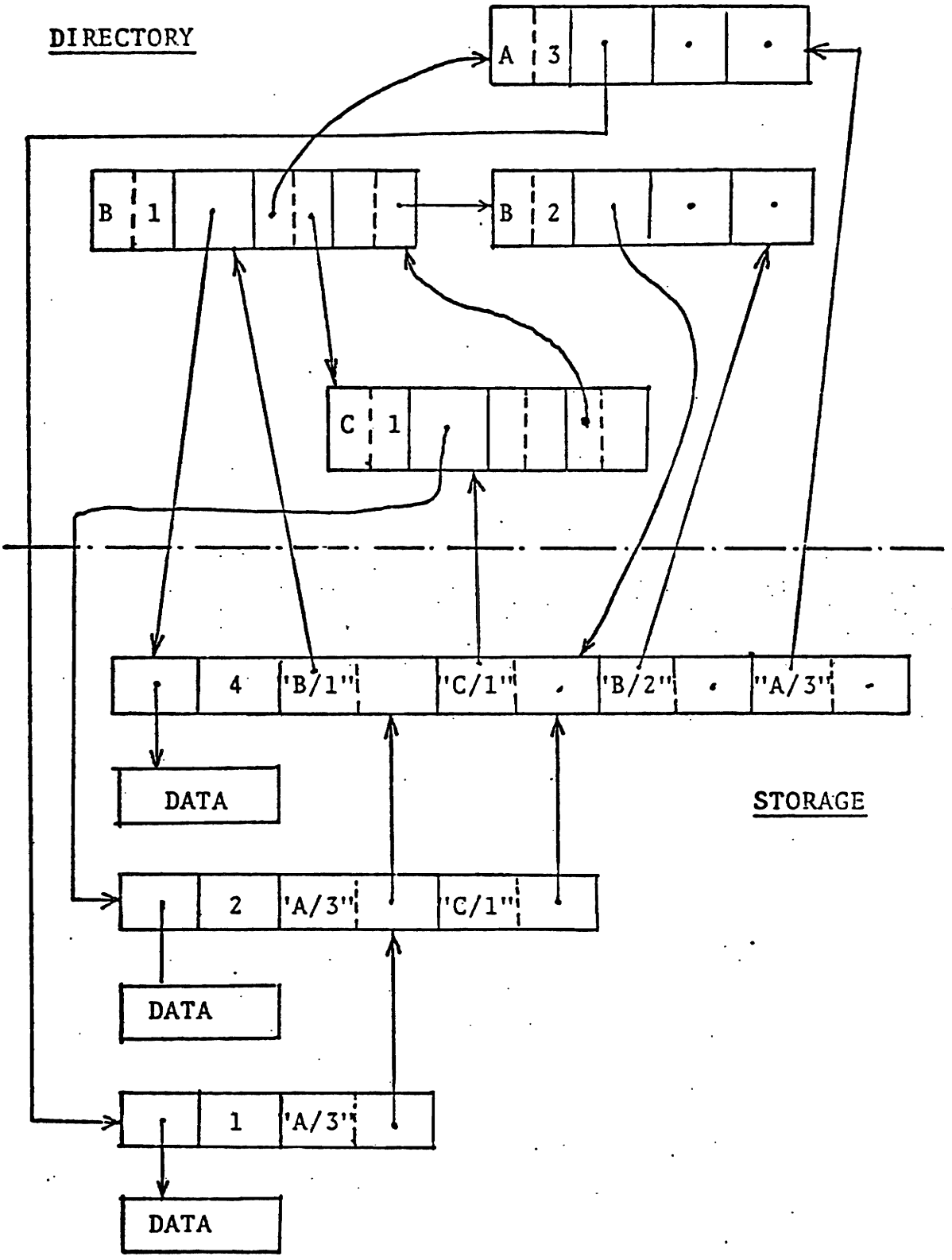


FIGURE 4.6 SAMPLE DIRECTORY AND STORAGE ENTRIES

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

4.4.2 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 4.4.1.

Algorithm 4.1 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 4.1 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.
```

Algorithm 4.1 (continued)

```
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).
```

Algorithm 4.1 (continued)

```
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.

4.4.3 RETRIEVE Routine

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 4.4.1 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 4.2 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 4.2 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.

ALGORITHM 4.2 (continued)

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 4.4.1). 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 4.3 outlines the steps of the procedure RETREVD. If option 0 is used, then the key type is

ALGORITHM 4.3 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.
```

ALGORITHM 4.3 (continued)

```
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).

4.5 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.

4.5.1 Source Specification Report

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 4.6). This report is useful to the user during the debugging stage. As shown in Figure 4.3, 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 4.7. This example will be followed through the remainder of this report.

4.5.2 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.

```
/* 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>  = OHMMETER(MRES OHM ))
                TARGET: MRES;

      ASSERTION $P_W0002($M_DC_INPUT):
          MRES >  100
                SOURCE: MRES;

    LOGIC $LOGIC0010(DC_INPUT): |~INP_SHORTED, *DISPLAY;

      DIAGNOSIS INP_SHORTED:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=INPUT_SHORT,
              TYPE=SHORTED_MSG;

      DIAGNOSIS DISPLAY:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(MRES, ' OHMS'),
              TYPE=TEXT;

  TEST AMPL;

    /* NULL STIMULI */

    MEASUREMENT $M_AMPL(AMPL);

      ASSERTION $P_W0001($M_AMPL):
          V_SIN =  0.26 +- 0.06
                SOURCE: V_SIN;
```

FIGURE 4.7: REFORMATTED NOPAL SOURCE SPECIFICATION FOR MINIRADIOSET. (THE ENTIRE SET OF REPORTS IS SHOWN IN APPENDIX A)

```
    LOGIC $LOGIC0010(AMPL): *SHOW_MEAS, |~FREQ_TOL;

      DIAGNOSIS SHOW_MEAS:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(V_SIN, 'VRMS'),
              TYPE=TEXT;

      DIAGNOSIS FREQ_TOL:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=AMPL_TOL(STD_5MHZ_FRE),
              OTHER PARAMETERS=('AMPLIFIER'),
              TYPE=FREQ_TOL_MSG;

TEST DISTORT_2W;

    STIMULI $S_DISTORT_2(DISTORT_2W);

       CONJUNCTION $S_W0001($S_DISTORT_2):
           (<J24_B, GND>  = PWR_SUPPLY(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(M_DISTORT % ,1 KHZ ))
                    TARGET: M_DISTORT;

       ASSERTION $M_W0002($M_DISTORT_2):
           M_DISTORT <= 5
                    SOURCE: M_DISTORT;

    LOGIC $LOGIC0010(DISTORT_2W): *DISTORT_PRNT, |~HI_DISTORTIO;

      DIAGNOSIS DISTORT_PRNT:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(M_DISTORT, '%'),
              TYPE=TEXT;

      DIAGNOSIS HI_DISTORTIO:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=DISTORT(AUDIO_2W),
              OTHER PARAMETERS=(' 2W', 5.0),
              TYPE=DISTORT_MSG;

TEST DISTORT_VOLT;

    STIMULI DCV_AMS(DISTORT_VOLT);

       CONJUNCTION $S_W0001(DCV_AMS):
           (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &
           (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

    MEASUREMENT $M_DISTORT_V(DISTORT_VOLT);
```

FIGURE 4.7: (continued)

```
        CONJUNCTION SM_W0001(SM_DISTORT_V):
            (<J19_A, GND>  = SINE_WAVE(VRMS VOLT ,*,0 SEC ))
                    TARGET: VRMS;

        ASSERTION A1(SM_DISTORT_V):
            VRMS >= 2.2
                    SOURCE: VRMS;

        ASSERTION A2(SM_DISTORT_V):
            VRMS <= 2.8
                    SOURCE: VRMS;

    LOGIC SLOGIC0010(DISTORT_VOLT): *WAIT, *VRMS_PRNT, |~VRMS_FAILED;

      DIAGNOSIS WAIT:
         OPERATOR MESSAGE:
             TYPE=TUNE_MSG,
             TIME= 0.00000E+00SEC,
          RESPONSE=?;

      DIAGNOSIS VRMS_PRNT:
         OPERATOR MESSAGE:
             OTHER PARAMETERS=(VRMS, ' VAC'),
             TYPE=TEXT;

      DIAGNOSIS VRMS_FAILED:
         OPERATOR MESSAGE:
             AFFECTED COMPONENTS=REF_VOLT(AUDIO_10MW),
             TYPE=VRMS_MSG;

TEST FREQ;

    STIMULI DCV(FREQ);

        CONJUNCTION SS_W0001(DCV):
            (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT ));

    MEASUREMENT SM_FREQ(FREQ);

        CONJUNCTION SM_W0001(SM_FREQ):
            (<J22, GND>  = SINE_WAVE(V_SIN VOLT ,FREQ HZ ,DELAY_TIME SEC ))
                    TARGET: FREQ, V_SIN
                    SOURCE: DELAY_TIME;

        ASSERTION SM_W0002(SM_FREQ):
            IF DELAY_TIME=60 THEN
                FREQ =  5E+06 +- 60
            ELSE
                FREQ =  5E+06 +- 2.5
                    SOURCE: FREQ, DELAY_TIME;

    LOGIC SLOGIC0010(FREQ): *GET_DELAY_TI, |~FREQ_TOL_FAI, *FREQ_PRNT;
```

FIGURE 4.7: (continued)

```
        DIAGNOSIS GET_DELAY_TI:
            OPERATOR MESSAGE:
                TYPE=WARMUP_MSG,
                TIME= 0.00000E+00SEC,
            RESPONSE=(DELAY_TIME);

        DIAGNOSIS FREQ_TOL_FAI:
            OPERATOR MESSAGE:
                AFFECTED COMPONENTS=FREQ_TOL(STD_5MHZ_FRE),
                OTHER PARAMETERS=('FREQ'),
                TYPE=FREQ_TOL_MSG;

        DIAGNOSIS FREQ_PRNT:
            OPERATOR MESSAGE:
                OTHER PARAMETERS=(FREQ, ' HZ'),
                TYPE=TEXT;

TEST DISTORT_10MW;

   STIMULI $S_DISTORT_1(DISTORT_10MW);

      CONJUNCTION $S_W0001($S_DISTORT_1):
          (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &
          (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

   MEASUREMENT $M_DISTORT_1(DISTORT_10MW);

      CONJUNCTION $M_W0001($M_DISTORT_1):
          (<J19_A, GND>  = DISTORTION(M_DISTORT % ,2 KHZ ))
                  TARGET: M_DISTORT;

      ASSERTION $M_W0002($M_DISTORT_1):
          M_DISTORT <= 3
                  SOURCE: M_DISTORT;

   LOGIC $LOGIC0010(DISTORT_10MW): *DISTORT_PRNT, |~AUDIO_DISTOR;

    /*** FOLLOWING DIAGNOSIS ALREADY DEFINED BEFORE:

      DIAGNOSIS DISTORT_PRNT:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(M_DISTORT, '%'),
              TYPE=TEXT;
                                                          ***/

      DIAGNOSIS AUDIO_DISTOR:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=DISTORT(AUDIO_10MW),
              OTHER PARAMETERS=(' 10MW', 1.0),
              TYPE=DISTORT_MSG;

   /*************************************************/
   /*                                               */
   /*  MESSAGES                                     */
```

FIGURE 4.7: (continued)

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


   MESSAGE SHORTED_MSG:
       TEXT='R/T DC INPUT SHORTED J24-B/J24-C ', 'AN/GRC-106 DEFECTIVE.  CHECK P
RINTOUTS FOR DEFECTS.', 'PRESS STOP.';

   MESSAGE TEXT:  ALIAS=DISPLAY,
       TEXT='(P1):   (P2) ';

   MESSAGE FREQ_TOL_MSG:
       TEXT='(C) DEFECTIVE.', '5.0 MHZ STD. OUT OF (P) TOLERANCE.';

   MESSAGE DISTORT_MSG:
       TEXT='(P1) AUDIO DISTORTION GREATER THAN (P2) PERCENT.';

   MESSAGE TUNE_MSG:
       TEXT='TUNE RECEIVER:  MC & KC CONTROLS TO 250000.', 'ADJUST AUDIO GAIN CO
NTROL FOR 2.2 TO 2.8 VAC', '(2.5 VAC NOMINAL).    PRESS  YES.';

   MESSAGE VRMS_MSG:
       TEXT='10 MW DISTORTION REFERENCE VOLTAGE FAILED.';

   MESSAGE WARMUP_MSG:
       TEXT='IF A 12 MINUTE WARMUP IS DESIRED, ENTER 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 00600: AUDIO_2W, FAILURE FUNCTION=DISTORT;

      /******************************************************/
      /*                                                    */
      /*  UUT CONNECTION POINTS                             */
      /*                                                    */
      /******************************************************/
```

FIGURE 4.7: (continued)

```
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      : J16, CONNECTOR=(COAXIAL, ),
        LIMIT=(UVOLT,  1.00000E+02,  0.00000E+00, GND),
        COMMENTS=' COAXIAL CABLE';

UUT_POINT      : J19_L, LIMIT=(VOLT,  7.00000E+01,  0.00000E+00, GND);

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: OHMMETER, FUNCTION TYPE=M, #PINS= 2,
        PARAM_01=(X, T, LIMIT=(OHM,  1.00000E+03,  1.00000E+00));

FUNCTION   120: AMPL_TOL, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    10: PWR_SUPPLY, FUNCTION TYPE=S, #PINS= 2,
        PARAM_01=(X, S, LIMIT=(VOLT,  6.00000E+01,  0.00000E+00));

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    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));

FUNCTION   140: DISTORT, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    30: SINE_WAVE, 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='AMPL., FREQ., TIME DELYD';

FUNCTION   130: REF_VOLT, FUNCTION TYPE=F,
```

FIGURE 4.7: (continued)

```
            PARAM_01=(COMPONENT, S);

      FUNCTION   110: FREQ_TOL, FUNCTION TYPE=F,
            PARAM_01=(COMPONENT, S);

        /*************************************************/
        /*                                               */
        /*  ATE CONNECTION POINTS                        */
        /*                                               */
        /*************************************************/


      ATE_POINT      : ATE_J24B, UUT_POINTS=(J24_B);

    END MINIRADIOSET;
```

FIGURE 4.7: (continued)

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).

4.5.3 Syntax Error/Warning Report

The Syntax error/warning 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 4.3 this report is generated by a collection of error message routines (see Section 4.3.2). 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 4.4. The warning messages indicate the assumptions made, or the corrective actions taken by the Processor. All messages issued during the different phases are collected in one file.

4.6. Cross Reference Reports

This section presents the subsystem which derives, checks the data types and produces a set of six reference reports as summarized in Table 4.1. As indicated in Figure 4.3, these reports are generated by a program module XREF (XREF1 and XREF2) whose input is the stored NOPAL specification. Section 4.6.1 describes the cross-reference and attribute report produces by XREF1. Section 4.6.2 presents the other cross-reference reports generated by XREF2.

4.6.0 Data Type Derivation and Checking

As well known, the use of data types in programming languages plays an important role in programming documentation and mistake prevention. In the NOPAL system, we used the ideas of "dynamic data type" and "derived data type" and decided to give the user some freedom in choosing the usage of data types.

The basic data types available are restricted by the types available in tho object language EQUETA ATLAS. The user of the NOPAL system can freely

decide what he wants to do about the data types, as long as he uses the variables consistently. That is, he may declare attributes for all the variables in his NOPAL program or declare certain data types for some variables or simply let the system derive all the data types according to the computation specified by the NOPAL assertions.

The basic data types (or attributes of variables) available in the NOPAL system are:

a) INTEGER (abbreviation: INT, code: 4);

b) DECIMAL (abbreviation: DEC, default, code: 1);

c) DIGITAL (abbreviation: DIG, code: 3);

d) BOOLEAN (abbreviation: VOL, 2-valued digital, code: 2).

Note that first, in NOPAL we do not handle the character string operation because the EQUATE ATLAS has no ability in handling this. Secondly, the code 0 is reserved for UNDEFINED DATA TYPE. We describe the implementation of the basic data type checking mechanism in the following.

The data type of a variable can be decided (or derived) by

1. DCL statements, i.e., DCL A digit;
   DCL B integer;...;
2. By the assignment containing the built-in function, i.e., if "A=TRUE" is an assertion then A is of type "dig" and if "I=SUBS('RPES',10);" is an assertion then tye type for I is "INT";
3. By associated operator within the scope of an assertion with most popular mathematical definition for that operator, i.e., if ASSR: A=B+C; then the data type of A, B and C will be of "int" or "dec" depending on the previous use of DCL statement(s);
4. By the assignment containing user-defined functions, i.e., the user has defined a function SINGEN and the "value" clause of the function definition is defined as "DEC", then if A(...)=SINGEN(...) is an assertion the data type of A is assigned as "DEC" as well;

5. By propagation, that is, if "A=B;", "B=C;" and "C=D;" are three assertions and if any one of the three variables has been decided to be of some type then the rest will be decided by propagating the given type.

The check for the use of data types if "operator based" (if we regard the functions as n-ary operators, with n the arity of the function), i.e., the data type of a variable is determined by the use of the variable in assertion. For example, if "A=B*C;" is an assertion, then the data types for B and C are decided by the use of the operator "*", i.e., they must be of type "INT" but if C is declared of type "DIG" or "BOL" which is incompatible with the definition of the operator, an error will be reported and the user is asked to modify the possible misuse of the variable or declaration statement. Finally, the data type for A is propagated from LHS and then of type "INT" or "DEC" depending on the declaration statements for B and C. If none of the variables are declared to be of any data type, the system will assume "reasonable" types for the variables according to the combination of variables and operators. This is why we called the data type check "operator based". Furthermore, the system tries to be as "reasonable" as possible in making the decisions, that is unknown types of variables will be given a type based on the above five criteria for assigning data types; if that effort fails, then "DEC" type will be assumed, because we think that the most generally used type is "DEC". The priority of assigning data types is:

1. Types declared by the user by using DCL statements;
2. Types defined by functions (built-in or user defined);
3. Types implied by corresponding operators;
4. Types propagated by assignment statements;
5. Default.

The table of correspondence between operators, built-in functions and data types is depicted as follows:

LHS(TYPE)	OPERATOR	RHS(TYPE)
"dec" or "int"	+,-,*,/,**	"dec" or "int"
"digtal"	"|,"&	"digital"
"digital"	"+,"*	"integer"
---------	"^	"digital"
"boolean"(s)	|,&	"boolean"(s)
---------	^(not)	"boolean"(s)
"dec" or "int"	<,<=, >,>=	"dec" or "int"
same as rhs	=,^=	same as lhs

BUILT-IN FUNCTIONS: SUBS() → "INT"
TRUE/FALSE → "BOOLEAN".

The implementation:

INPUT: User-specified NOPAL statements (stored in associated memory);

OUTPUT: Array DATATYPES(MSDIRS).

Algorithm:

1) Initialization for DATATYPES;

2) Derive the data types defined by data declaration statements;

3) Derive the data types implied by the use of NOPAL built-in functions;

4) Start the derivation of data types implied by the operators by using two recursive routines for IF-clauses and simple assertions respectively;

5) The collected unknown type variables were entered into TYPESTACK and the propagation of the types among the unknowns is taken place iteratively until there is no further possible derivation;

6) Fill up the rest of the DATATYPES with Code4 (meaning decimal).

Finally, the derived types are passed to XREF1 procedure and printed in cross-reference report.

4.6.1 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 4.8 shows the cross-reference and attribute report from the example given in Figure 4.7.

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 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).

CROSS REFERENCE AND ATTRIBUTES REPORT

NAME	DEF NO.	ATTRIBUTES AND REFERENCES	DATA TYPE
A1	29	ASSERTION LABEL	———
A2	30	ASSERTION LABEL	———
AMPL	9	TEST LABEL 10 12	———
AMPL_TOL	79	ATE-FUNCTION ID , M 14 69	DECIMAL
ATE_J24B	82	ATE-POINT ID	———
AUDIO_10MW	70	COMPONENT ID , WITH FAILURE-FUNCTION: REF_VOLT 34	———
AUDIO_10MW	71	COMPONENT ID , WITH FAILURE-FUNCTION: DISTORT 52	———
AUDIO_2W	72	COMPONENT ID , WITH FAILURE-FUNCTION: DISTORT 23	———
CONST_R	74	ATE-FUNCTION ID , M 4	DIGITAL
CONST_S	73	ATE-FUNCTION ID , S 37 26 17 46	DIGITAL
D	53	MESSAGE LABEL 8 13 22 33 44 51	———
D2	7	DIAGNOSIS LABEL 6	———
D3	8	DIAGNOSIS LABEL	———
D4	42	DIAGNOSIS LABEL 41	———
D5	43	DIAGNOSIS LABEL 41	———
D6	44	DIAGNOSIS LABEL 41	———
D7	13	DIAGNOSIS LABEL 12	———
D8	14	DIAGNOSIS LABEL 12	———
DCV	36	STIMULUS LABEL 37 26	———
DCV_AMS	25	STIMULUS LABEL 26 17 46	———
DC_INPUT	2	TEST LABEL 3 6	———
DISLPAY	53	SYNONYM OF MESSAGE LABEL : D	———
DISTORT	81	ATE-FUNCTION ID , M 23 52 71 72	DECIMAL
DISTORTION	76	ATE-FUNCTION ID , M 19 48	INTEGER
DISTORT_10MW	45	TEST LABEL 46 47 50	———
DISTORT_2W	15	TEST LABEL 16 18 21	———
DISTORT_VOLT	24	TEST LABEL 25 27 31	———
F1	39	VARIABLE ID 40 44	DECIMAL

Figure 4.8 First page of cross reference report for MINIRADIOSET.

Any errors or warnings which are detected during this phase of cross-referencing are printed in a separate cross-reference error report. A complete list of the messages are given in Table 4.3.

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. Third, the dimensions of variables occurring in structure declarations are propagated to their decendents.

As described in greater detail in Chapter 5, the variables that are global (or local) to a test module are involved in determining one type of precedence relationship, called data determinancy, in the phases of 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 analysis and sequencing of the test modules is started. If a variable x has ever been defined as TARGET alone, or used as SOURCE alone in any test module, or has a declaration then X is a global variable (i.e. has global scope with respect to the test modules. Otherwise, X is local (i.e. has local scope with respect to each test module where it has been both defined as TARGET and used as SOURCE).

Algorithm 4.4 determines the scope of every variable in the specification. It also designates, as SOURCE, every variable

ALGORITHM 4.4 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 an operator response variable. go to S9.

S6.1 If x is data declaration then
(a) set scope as global
(b) go to S9

S6.2 If x is parameter in test module then
(a) set scope as local
(b) got 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.
```

ALGORITHM 4.4 (continued)

```
S12:  Set 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 S14.1.
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:

        S14.1  If STMT.TYPE is not waveform then go
        to S15.1
S15:  Set     N = 1;
      Set TEST_IDS(1) = the test label of the
          waveform; Go to S17;

        S15.1  If STMT.TYPE is not test or module
               function go to S17
        S15.2  N = 1
               Set TEST_IDS(1) = the label of test
               or module function
               Go to S17;
        S15.3  If STMT TYPE is not data declaration
               then go to S17.
        S15.4  N = 0;/* Not part of any test or module
               function*/
               ND = ND + 1
               DEFN(ND) = TVAL (I)
               REFL (ND) = #REF
               Go to 17
```

ALGORITHM 4.4 (continued)

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

S18: Set TEST_ID = TEST_ IDS(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.

ALGORITHM 4.4 (continued)

```
            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 do S32.
      S32:  /* some residual        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 SO */
     S46: Return;
```

which has not been explicitly declared either as SOURCE or TARGET in the assertions. This relieves the user from explicitly declaring such variable as SOURCE.

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.

4.6.2 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). These 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 4.1). These reports provide better man-machine interface and give the user a clear picture of inter-actions 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. (Figure 4.9)

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. (Figure 4.10).

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

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

Figure 4.9 Diagnosis-Test Cross Reference Report
For MINIRADIOSET

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

MESSAGE	DIAGNOSES	TEST-MODULES
#4	D2,	DC_INPUT,
D/DISLPAY	D3, D7, 227, 25, D6, 27,	AMPL, DISTORT_2W, DISTORT_VOLT, FREQ, DISTORT_10MW,
#6	D8, D5,	AMPL, FREQ,
#18	30, 28,	DISTORT_2W, DISTORT_10MW,
#15	24,	DISTORT_VOLT,
#17	26,	DISTORT_VOLT,
#5	D4,	FREQ,

Figure 4.10 Message-Diagnosis-Test Cross Reference Report For MINIRADIOSET

The COMP-DIAG-TEST reports lists all the affected components (i.e., component failures), with all the referencing diagnoses and test modules. (Figure 4.11).

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 (Figure 4.12).

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 4.13).

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 --- 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,
00600: DISTORT(AUDIO_2W)	30,	DISTORT_2W,

Figure 4.9 Affected Component-Diagnosis-Test Cross Reference Report For MINIRADIOSET

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),	

Figure 4.12 UUT Connect Point-Test-ATE Connection PTS
Cross Reference Report For MINIRADIOSET

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),

Figure 4.13 ATE Function Cross Reference Report
For MINIRADIOSET

CHAPTER 5

SPECIFICATION ANALYSIS AND SEQUENCE DETERMINATION

5.1 Overview of Subphases

This phase of the NOPAL processor deals with the analysis of a NOPAL specification and determination of the sequence of events in the program. 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 4.2 will be frequently referred to in the subsequent discussions.

In a NOPAL specification, each entity (e.g. modfun, 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.

The types of precedence relationships dictate the following: 1) how the conjunction and assertions in a test 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 computed in a predecessor test module before it can be used in a successor test module. This kind of precedence information can be represented by a directed graph. There is one separate graph for each one of the test modules and one for the entire collection of test modules in a

modfun. (Note that the present implementation allows one test per modfun in a MODULE, only the MAIN MODULE may have any number of tests in its modfun.) A directed graph consists of nodes and edges. Nodes of this graph represent the entities form the NOPAL specification such as variables, tests, and diagnoses. An edge shows that there is a relationship between a pair of nodes and the label of the edge identifies this relationship. If there is no edge between two nodes it means that they have no immediate relationship.

A precedence relationship indicates that one node must precede the other at execution time of the object test program. Thus one entity is said to be a predecessor of the other, 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 5.1. There are only five relationships within a test module: data determinancy, waveform setup, hierarchichal, pointer, and waveform diagnosis relationships. These are further explained later.

A directed graph (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. The weighted digraph shown in Figure 5.1 corresponds to the inter-test-module relationships of the example of Figure 4.7. In this graph each node represents the name of one of the entities in the NOPAL specification: test modules, diagnoses, and (global) variables. Note that each node may have 0, 1, or more edges emanating from it to successor nodes. The labels of the edges can be interpreted using Table 5.1.

Although a pictorial representation of a graph is convenient for visual study, there are other representations which are better suited for computer processing. The representation called adjacency matrix is used in the analysis phase of the NOPAL processor.

Given a digraph G = < N, E > consisting of a set of n nodes $N = \{V_1, V_2, \ldots, V_n\}$ and a set of edges E, an adjacency matrix, A, corresponding to the digraph G is an n by n matrix such that for all i and j (i,j = 1,2, ... n)

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

$= 0$ otherwise.

TABLE 5.1 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 with global TARGET variable X, (b) Global variable X, (c) Diagnosis with global operator response variable X	(a) Variable X (b) Test module using X as SOURCE		Global variable is evaluated in predecessor or referenced in successor
2	1	Inter-activeness	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 proper subset of D's	D is selected	If D asserts more generic failures, then more specific tests are conducted
6	2		Diagnosis D whose affected components are in conjuction	Test module whose affected components set is a	D is not selected	If D isolates some faults, then skip tests for subset of the same faults
9	3	Stimuli application	Test module which has a stimulus triplet which is globally more frequent	Test module which has a stimulus conjunction with is globally less frequent.		Once a stimuli is applied, as many tests as possible are performed.

TABLE 5.1 (continued)

Precedence			Relationship selection rule		Run-time condition	Explanation
Type	Priority	Strategy name	PREDECESSOR	SUCCESSOR		
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 through 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"(&¬)		

A weighted adjacency matrix 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 integer indicating the type of relationship whenever the corresponding entry in A has a value of 1. A weighted adjacency matrix corresponding to the digraph G is an n by n matrix W such that for all i and j (i,j = 1,2 ... n)

W(i,j) = k iff(V_i, V_j) is an edge of type k in E, i.e. (V_i, V_j) is a relation of type k;

= 0 otherwise

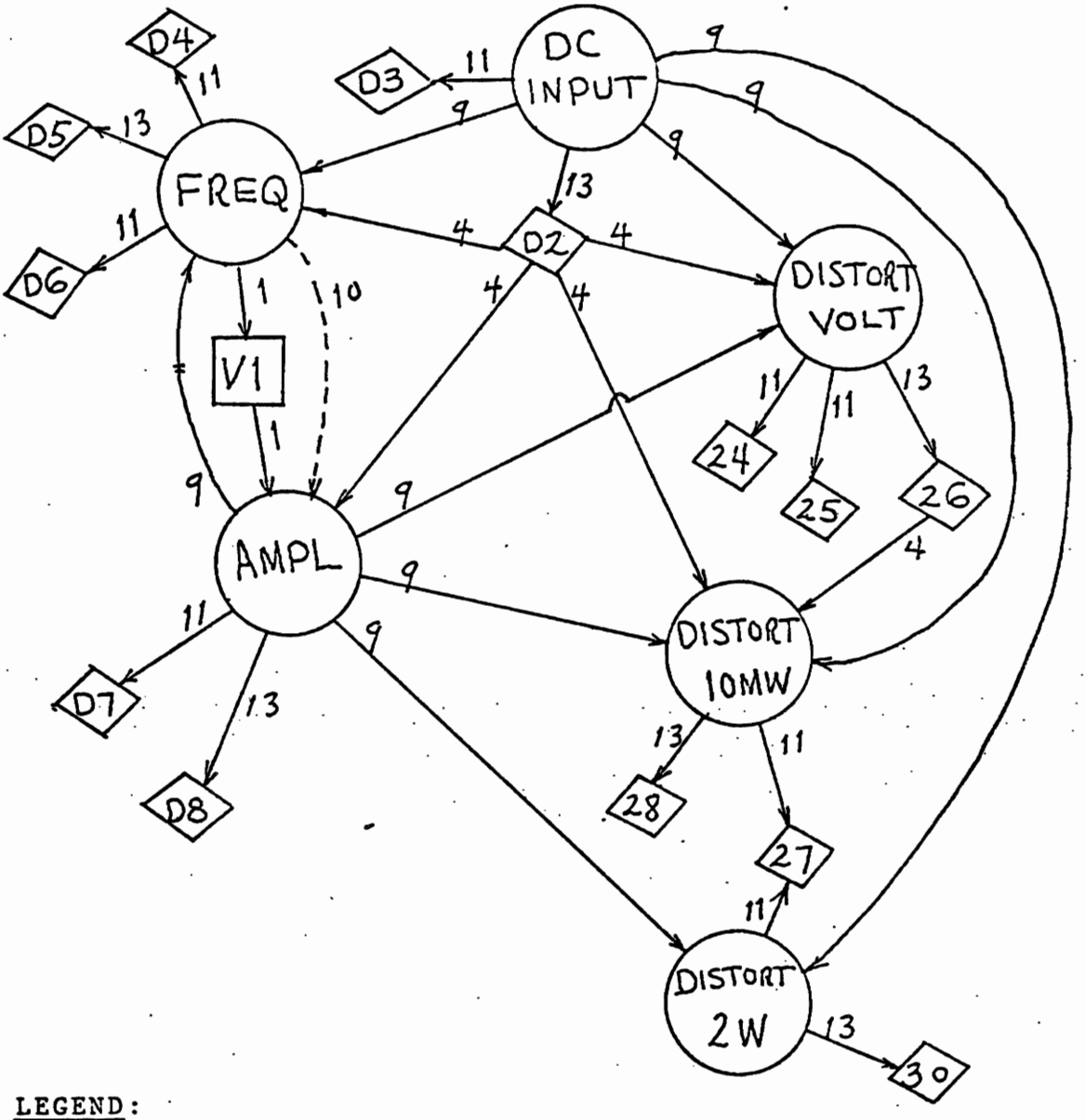


Figure 5.1 Digraph for NOPAL Specification MINIRADIOSET

The weighted adjacency matrix for a NOPAL specification is used extensively in the phases of analysis and sequencing. The matrix corresponding to the MINIRADIOSET example of Figure 4.7 (hence the digraph of Figure 5.1) is shown in Figure 5.2. The node numbers to the left of the node names are assigned by the Processor. Any entry (i,j) in the matrix is either 0, indicating that no relationship exists between node i (at i-th row) and node j (at j-th column), or an integer corresponding to the type of precedence relationships between node i and node j. These type numbers correspond to the precedence types listed in Table 5.1. The precedence types and relationships which are used from the sample specification MINIRADIOSET are enumerated in Table 5.2.

All the global precedence information is conveyed by a weighted adjacency matrix for the whole NOPAL specification. 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 5.3, there are two major subphases of analysis and sequencing: (1) intra-test-module. and (2) inter-test-module. The first subphase deals with only the waveform conjunctions, assertions, and diagnoses of a given test module. In Section 5.2 the subphases of intra-test-module analysis and sequence determination are presented. The second subphase deals with the whole collection of test modules in a given NOPAL specification, considering each test module as an integral unit. This phase is needed only for a MAIN MODULE or SPECIFICATION.

			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 5.2 WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION MINIRADIOSET

TABLE 5.2 ILLUSTRATION OF PRECEDENCE RELATIONSHIPS FOR MATRIX OF FIGURE 5.2

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_8,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.

In Section 5.3 the subphases of the inter-test-module analysis and sequencing are presented.

A weighted digraph of the NOPAL specification as represented by a weighted adjacency matrix is 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. As these conditions are found, they are printed either as warning or error messages. Further error analysis occurs after the matrix has been constructed. Table 5.3 summarizes the error and warning messages which can be detected 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.

In addition to the above messages, a number of error and warning messages can be issued during subscript analysis. Even though the user may submit a test specification which is correct according to the EBNF syntax of the language, the semantics associated with subscripted variables, test points, and some reserved evaluation functions may indicate erroneous usage. The errors and warning emenating from such cases are detected during the processing of the subscripts in the intra-test-module analysis phase. The context of these messages, and a brief explanation for each case is given in a separate table (Table 5.4)

Table 5.5 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 appli-

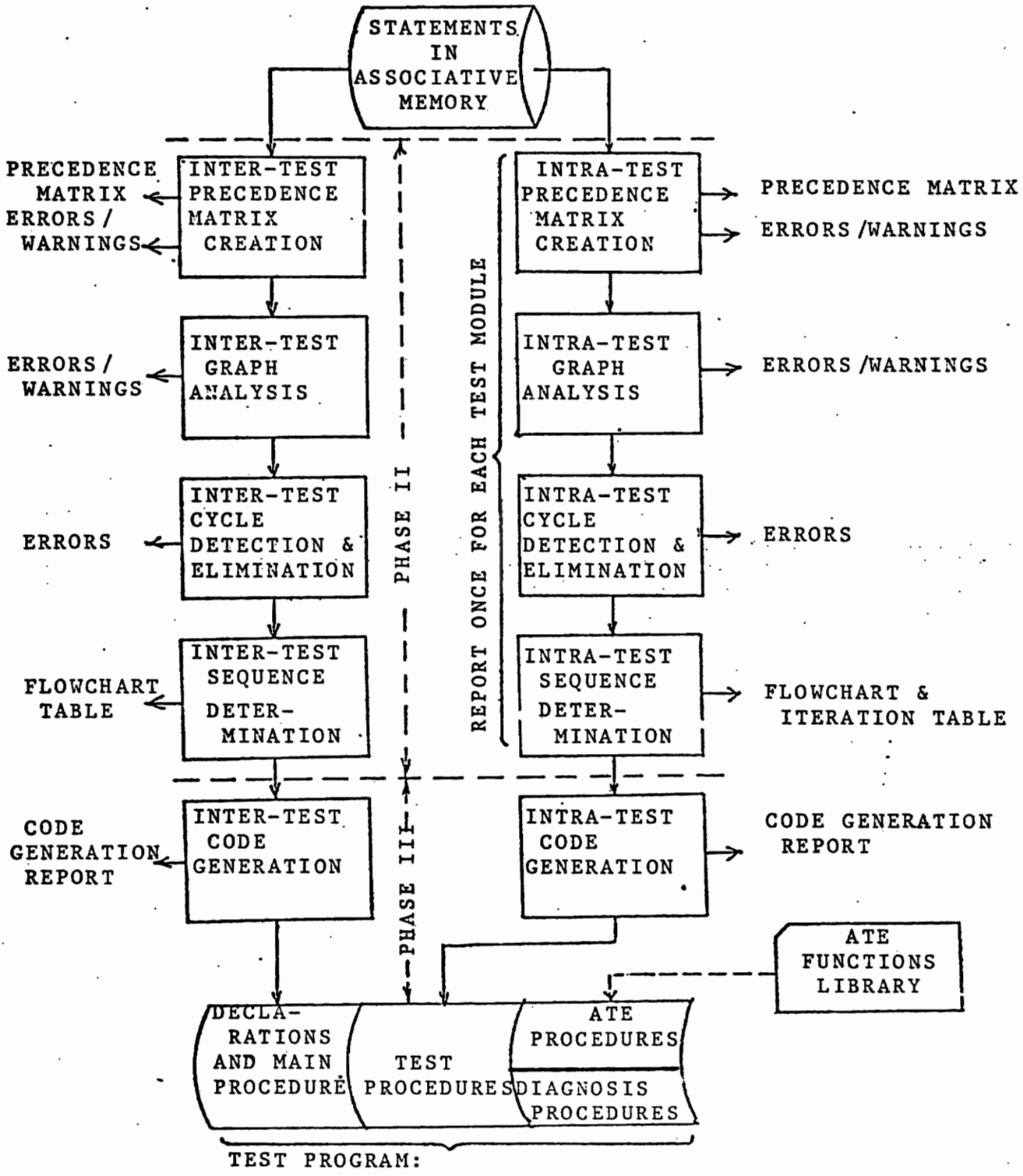


FIGURE 5.3 FLOWCHART FOR PHASES II AND III OF NOPAL PROCESSOR

cable 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 Section 5.2 and 5.3, respectively.

TABLE 5.3

ERROR/WARNING MESSAGES (XREF/ANALYSIS)

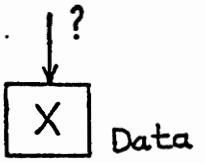
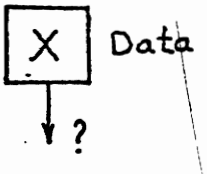
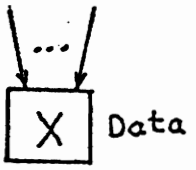
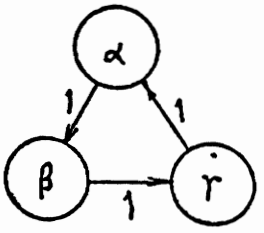
Ref#	Message	Issued by	Brief explanation/Example
1	ERROR (incompleteness): Variable X is used as SOURCE in ...; but its target definition never given elsewhere.	EXTSEQ	
2	WARNING (possible incompleteness): Variable X is defined as TARGET in ...; but never used elsewhere	EXTSEQ	
3	WARNING (possible ambiguity): 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 5.3

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

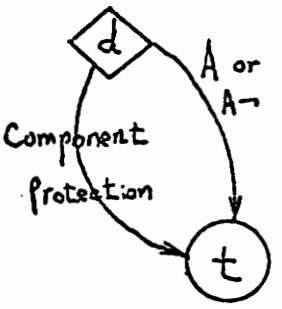
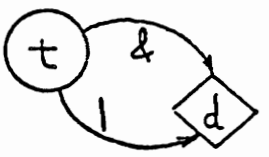
Ref#	Message	Issued by:	Brief explanation/Example
6	ERROR (ambiguity): TARGET variable x in assertion y of test z is <u>not</u> the only expression at the left-hand side of the equal sign ("=")	INTSEQ	W = X + 1 TARGET: X; <u>or</u> X - 1 = W TARGET: X;
7	WARNING (Inconsistency): In assertion x of test 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 5.3

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

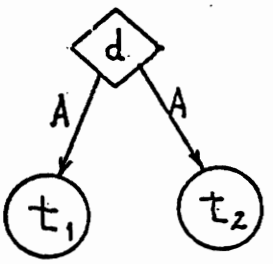
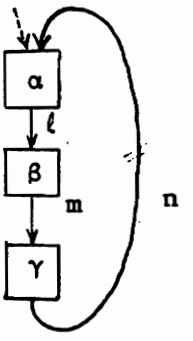
Ref#	Message	Issued by:	Brief explanation/Example
11	WARNING (possible ambiguity): X is defined/used as a local variable in tests...; and also in diagnoses ...	XREF1	Local variable X used in two diagnoses of two different tests.
12	WARNING (apparent inconsistency): In stmnt xxxx, X multiply defined. 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	
16	WARNING (circular definition): The following groups of items α,β,γ... were circularly defined. The cycle was eliminated by deleting item n from the weighted adjacency matrix.	CYCLES	

TABLE 5.4: ERROR AND WARNING MESSAGES FROM SUBSCRIPT ANALYSIS

#(1)

Message:

ERROR(INCONSISTENT):

In test t, the number of dimensions for variable X is inconsistently defined.

When Issued/Examples:

```
Y = A(3,5);
A(I) = X(I)          TARGET:  A(I);
```

The first statement uses 'A' as a two-dimensional array while the second statement uses 'A' as a one dimensional array.

Issued by routine: SUBANAL

#(2)

Message:

WARNING(INCOMPLETE):

Dimension X of variable Y is not declared in test Z, bound undecided.

When Issued Examples:

```
I = SUBS('A:1',10)          TARGET:  I;
K = SUBS('A:3',5)           TARGET:  K;
A(I,J,K)=0                  TARGET:  A(I,J,K);
```

If the second dimension of 'A' has never been declared but is used (e.g. in the form of A(I,J,K)), then this message is issued.

Issued by routine: SUBANAL.

(3)

Message:

ERROR(Ambiguous): In statement number N ---

Subscript declaration is not a simple assertion.

When Issued/Examples:

```
I = SUBS('A:1', 10) + 20              TARGET:  I;
I <= SUBS ('A',5);
```

TABLE 5.4: (continued)

The first statement asserts that 'SUBS' is an arithmetic function that returns a value. The second statement is not an explicit-relational-type assertion.

Issued by routine: PARSE_SUB

(4)

Message:

ERROR(Ambiguous): In Statement Number N ---

Excessive blank appears in the parent list.

When Issued/Examples:

I = SUBS('A B', 10) TARGET: I;

Issued by routine: PARSE_SUB

(5)

Message:

ERROR(Incomplete): In Statement Number N ---

The parent list of subscript declaration is missing or not enclosed in quotes.

When Issued/Examples:

I - SUBS(A,B,10) TARGET: I;

I = SUBS(10) TARGET: I;

The first statement is missing quotes around the parent list. The second statement is missing the parent list.

Issued by routine: PARSE_SUB

(6)

Message:

ERROR(Ambiguous): In Statement Number N ---

Improper use of comma in the parent list.

TABLE 5.4 (continued)

When Issued/Examples

I = SUBS(',A:1',10) TARGET: I;

J = SUBS('A,,B:1' 5) TARGET: J;

Issued by routine: PARSE_SUB

(7)

Message:

ERROR(Incomplete): In statement number N ...

Null parent list is used.

When Issued/Examples:

I = SUBS ('',10) TARGET: I;

Issued by routine: PARSE_SUB

(8)

Message:

ERROR(Ambiguous): In statement number N ...

The dimension number is not a positive integer.

When Issued/Examples

I = SUBS('A:5H',10) TARGET: I;

J = SUBS('A:N',5) TARGET: J;

Issued by routine: PARSE_SUB

(9)

Message:

ERROR(Ambiguous): In statement number N ...

Improper use of colon.

When Issued/Examples:

I = SUBS('A: :2', 10) TARGET: I;

Issued by routine: PARSE_SUB.

TABLE 5.4 (continued)

(10)

Message:

ERROR(Inconsistent): In statement number N ...

The upper bound is not a positive integer or *.

When Issued/Examples

```
I = SUBS('A',B)                    TARGET: I;
J = SUBS('A',-4)                   TARGET: J;
```

Issued by routine: PARSE_SUB

(11)

Message:

ERROR(Inconsistent): In statement number N ...

Variable X is not used as a subscripted variable in the test.

When Issued/Examples:

```
I = SUBS('A',10)          TARGET: I;
A = 0                     TARGET: A;
```

First statement declares A as a subscripted variable. However, A is used as a scalar in the test as the second assertion indicates.

Issued by routine: PARSE_SUB

(12)

Message:

ERROR(Inconsistent): In statement number N ...

Dimension X of variable Y is multiply defined.

When Issued/Examples:

```
I = SUBS('A:2', 10)          TARGET: I;
J = SUBS('B,A:2',5)          TARGET: J;
```

Issued by routine: PARSE_SUB

TABLE 5.4 (continued)

#(13)

Message:

WARNING(Ambiguous): In statement number N ...

Parent name X has never appeared in the test module.

When Issued/Examples:

I = SUBS('A',10) TARGET: I;

When A is declared as a subscripted variable but is never referenced in the test module.

Issued by routine: PARSE_SUB

(14)

Message:

ERROR(Incomplete): In statement number N ...

Subscript declaration is not correct, check use.

When Issued/Examples:

I = SUBS('A') TARGET: I;

When the subscript is improperly declared in addition to the previous cases (i.e. messages 3 to 13), this general message is issued. In the example, the declaration is incomplete because the bound field is missing.

Issued by routine: PARSE_SUB

#(15)

Message:

ERROR(Ambiguous): in statement number ---

Variable X is multiply declared as free subscript.

When Issued/Examples:

When a variable is declared more than once within the same test module, then we issue the above message.

e.g.

I = SUBS('A:1', 10) TARGET: I;

TABLE 5.4 (continued)

I = SUBS('A:2', 5) TARGET: I;

Issued by routine: PARSE_SUB

(16)

Message:

ERROR(Inconsistent): in statement number N ...

The free subscript X in source variable does not appear as free subscript in target variables.

When Issued/Examples:

A = B(I) TARGET: A SOURCE: B(I), I;

When I is declared as a free subscript, the above assertion will repeat many times. Then an error occurs since A is multiply assigned a value.

Issued by routine: SUBUSAG.

(17)

Message:

ERROR(Inconsistent): in statement number N ...

Subscript in variable X does not follow the correct syntax.

When Issued/Examples:

C = D(3.5) TARGET: C SOURCE: D(3.5);

The assertion uses a non-integer '3.5' as subscript. Thus the above error message is issued.

Issued by routine: SUBUSAG

(18)

Message:

WARNING(Possible inconsistent): in statement number N ---

I-th subscript of variable X is a subscripted variable or a non-free subscript; range test is not performed.

TABLE 5.4 (continued)

When Issued/Examples:

A(I) = B(C(I)) TARGET: A(I) SOURCE: B(C(I)), C(I), I;

A(I) < B(X) SOURCE: A(I), B(X), I, X;

Suppose B is declared as a vector of size 10. Since we do not know the value of C(I) and X until run time, range test cannot be performed at compile time.

Issued by routine: SUBUSAG.

(19)

Message:

ERROR(Inconsistent): in statement number N ---

Variable X is not found in the variable table or dimension-I is not defined.

When Issued/Examples:

This message is isssued when a variable is not declared as being subscripted (through subscript declaration statement) and is used. For example, if A is not declared as a vector, then

A(5) > 0 SOURCE: A(5)

Causes an error.

Issued by routine: SUBUSAG.

(20)

Message:

ERROR(Inconsistent): in statement number N ---

The subscript bound of variable X for dimension-I was declared to be M, but N is used here.

When Issued/Examples:

When the actual subscript used is greater than the upper bound declared, the above message is issued.

TABLE 5.4 (continued)

I = SUBS('A:1', 5) TARGET: I;

A(10) > 5 SOURCE: A(10);

Issued by routine: SUBUSAG

#(21)

Message:

ERROR(Ambiguous): in statement number N ... .

Reduction function is not used in a simple explicit assertion.

When Issued/Examples

X < SUM(A(I),I) SOURCE: X,A(I),I;

IF VAR > 60 THEN Y = SUM(A(I),I) ELSE Y=SUM(B(I),I) TARGET: Y;

The first statement is not an explicit relation assertion. The second statement is a conditional assertion.

Issued by routine: REDUSAG.

(22)

Message:

ERROR(Ambiguous): in statement number N ...

Instead of 2, P arguments are used in the reduction function.

When Issued/Examples:

X = SUM(A(I), B(I), I) TARGET: X;

Y = SUM(C(I,J)) TARGET: Y;

The reduction function (e.g. SUM) should always have two arguments. The first statement has three arguments while the second statement has one.

Issued by routine: REDUSAG

(23)

Message:

ERROR(Ambiguous): in statement number N ...

The second argument X of the reduction function is not declared as a free subscript.

TABLE 5.4 (continued)

When Issued/Examples:

```
X = SUM(A(I), B(I))                    TARGET:  X;
Y = SUM(A(I),10)                       TARGET:  Y;
```

Issued by routine: REDUSAG

(24)

Message:

ERROR(Ambiguous): in statement number N ...

The first argument X of the reduction function is not a subscripted variable.

When Issued/Examples:

```
X = SUM(A,I)                           TARGET:  X;
Y = SUM((SUM(I,J),J),I)                TARGET:  Y;
```

First statement has 'A' a simple variable, as the first argument of a reduction function. Second statement uses a function as the first argument.

Issued by routine: REDUSAG

(25)

Message:

ERROR(Ambiguous): in statement number N ...

The I-th subscript X of variable Y is not a simple variable or constant.

When Issued/Examples:

```
X = SUM(A(C(I)),I)                     TARGET:  X;
Y = SUM(A(I,J+2),I)                    TARGET:  Y;
```

The first statement uses C(I) as subscript while the second statement uses J+2 as subscript.

Issued by routine: REDUSAG

TABLE 5.4 (continued)

#(26)

Message:

ERROR(Ambiguous): in statement number N ...

More than one dimension may be reduced in a reduction function.

When Issued/Examples:

X=SUM(A(I,J,I),I) TARGET: I;

'I' is the first and third subscripts of variable 'A'; we do not know which dimension is to be reduced.

Issued by routine: REDUSAG

#(27)

Message:

ERROR(Ambiguous): in statement number N ...

The subscript being reduced X is not found in the first argument Y.

When Issued/Examples:

X=SUM(A(I),J) TARGET: X;

Y=SUM(A(I,J),K) TARGET: Y;

This message is issued when the second argument does not appear in the subscript list of first argument; we do not know which dimension is to be reduced.

Issued by routine: REDUSAG

#(28)

Message:

ERROR(Inconsistent): in statement number N ...

The dimensionality of LHS is not one less than that of RHS.

When Issued/Examples:

X(I) = SUM(A(I,J,K),J) TARGET: X(I);

Y = SUM(A(I,J),I) TARGET: Y;

Issued by routine: REDUSAG.

TABLE 5.4 (continued)

#(29)

Message:

ERROR(Ambiguous): in statement number N ...

The LHS X of a reduction function is not a subscripted variable or scalar.

When Issued/Examples:

SIN(X) = SUM(A(I),I)

When LHS of an assertion is not a (subscripted) variable, this message is issued.

Issued by routine: REDUSAG.

#(30)

Message:

ERROR(Inconsistent): in statement number N ...

Subscript X in left-hand side of a reduction function does not appear in right-hand side.

When Issued/Examples:

A(I) = SUM(B(I,J),I) TARGET: A(I);

Subscript I is reduced and should disappear from subscript list of LHS. Instead, A(J) should have been the target variable.

Issued by routine: REDUSAG

#(31)

Message:

ERROR(Ambiguous): In statement number N ...

Reduction function is improperly used.

When Issued/Examples:

When the usage of a reduction function is incorrect in addition to the reasons cited before (messages 21 to 30), this general message is issued.

TABLE 5.4 (continued)

X = SUM(A(I),I)+5 TARGET: X;

Y = 2*SUM(A(I),I) TARGET: Y;

Both of the above examples include an arithmetic operator in RHS of the assertion.

Issued by routine: REDUSAG.

#(32)

Message:

ERROR(Incomplete): in test X

Not all nodes are rankable, return from PRECED.

When Issued/Examples:

When some nodes are not rankable due to circular definition or incorrect usage of subscripts, this message is output.

Issued by routine: SCHEDLER

#(33)

Message:

ERROR(Inconsistent):

Global variable V is used as an array, but with different number of dimensions. Initially used with DIM# (i) number of dimensions.

When Issued/Examples:

A(I,J) < 5 SOURCE: A(I,J);

A(I) = VAR TARGET: A(I);

Global variable A is used as both 2-dimensional and 1-dimensional array: then this message is issued.

Issued by routine: EXTSEQ

TABLE 5.4 (continued)

#(34)

Message:

ERROR(Inconsistent):

Global variable V is used as an array and scalar, it cannot be both.

When Issued/Examples:

```
A = 5           TARGET: A;
A(I) > 6        SOURCE: A(I);
```

Global variable A is used as both a scalar and an array. Then this message is issued.

Issued by routine: EXTSEQ

TABLE 5.5 SUMMARY OF 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 an n by n 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.	Subscript Analysis	Check the syntax and semantics of the usage of subscripted variables.
5.	Cycle Detection and Elimination	Create path matrix from A; search for possible cycles; delete the cycles if possible; otherwise report as error to user.
6	Sequence Determination	Rank the nodes of the digraph according to precedence, and then reorder the nodes by their rank.

5.2 Intra-Test-Module Analysis and Sequencing

5.2.1 Overview of Intra-Test-Module Analysis

This section provides in greater detail the subphases of inter-test-module analysis. These subphases as shown in Figure 5.4 include the graph creation, analysis, subscript processing, and sequencing. The overall program which performs intra-test-module analysis and sequencing is named INTSEQ. INTSEQ generates, as output, a vector, ORDER, of nodes ordered in their execution sequence and an iteration table, DOTAB, which identifies the iteration variables and iteration scopes. INTSEQ also generates messages if errors are detected in each subphase. The error and warning messages issued by the first four lines of Figure 5.4 are referred to by the reference numbers given in Table 5.3, while the messages issued by subscript analysis, subscript propagation, and sequencing are listed in Table 5.4.

Section 5.2.2 discusses the creation of the graph (in adjacency matrix form) and the entering of precedence relationships. Section 5.2.3 deals with preliminary analysis of the graph. Section 5.2.4 discusses the subjects of cycle detection, enumeration, and elimination. The processing of subscript statements is described in detail in Section 5.2.5. Section 5.2.6 presents the method of subscript propagation, and section 5.2.7 discusses the rationale and methodology of determining execution sequence. An example of a flowchart report as well as a brief explanation of the report are provided in section 5.2.8.

Note that each subphase in INTSEQ must be repeated once for each test module in the whole NOPAL test specification.

Since the current implementation allows only one test per modfun in a MODULE, sequencing this test implies sequencing the modfun.

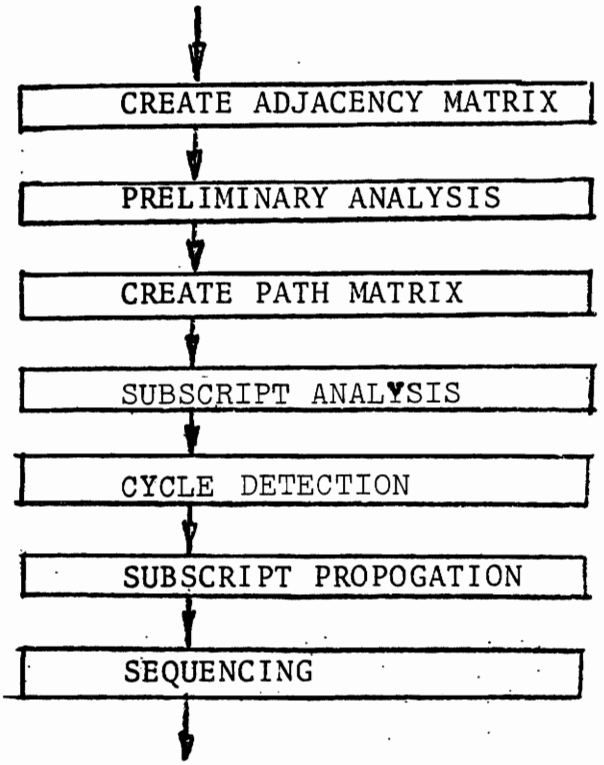


FIGURE 5.4 FLOWCHART OF INTRA-TEST MODULE ANALYSIS AND SEQUENCING

5.2.2 Creation of Weighted-Adjacency Matrix

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 5.1 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, assertions, diagnoses, and the variables in the test module. Evaluation and control functions are treated as global variables which never have to be targeted, i.e. they are always available as global source variables.

Five 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.

Hierarchichal relationships between the variables may exist due to the declaration of structures. Such relationships are entered as being of type 19. A parent node in the structure declarations is entered as the predecessor of its children in the structure if the structure is of type input. For output structures the precedence relationship is reversed. A structure is considered to be of type

input if its leaves are source variables and are never targetted to.

Pointer relationship exists between a variable having the prefix 'PTR_' and the structure named as the suffix of the variable (or 'PTRy_', where y is a digit from 1 to 5). For example: PTR_X is related to the structure x by means of pointer relationship. This is entered as type 20.

The data determinacy (type 1), the hierarchichal relationship (type 19) and pointer relationship (type 20) are all mandatory, hence they are associated with the highest priority 1. The waveform setup relationship is implied only if the measurement conjunction does not generate a variable which in turn is used in the stimulus conjunction. Therefore, this relationship is associated with a lower priority 2 and is removed in case of conflict with an edge of priority 1. This conflict appears as a cycle in the graph and is removed by the CYCLES algorithm.

Relationships between the waveforms and diagnoses are also entered based on the logic. These are put in to express the idea that * diagnoses (which are select unconditionally by the logic) should be performed before the waveforms; all other diagnoses (selected by &, &¬, |, |¬) should be selected after the waveforms. However, these have low priority (=5) associated with them and may be deleted by the cycle elimination algorithm in case a cycle is formed because of their presence. To give the actual types: type 16 is entered between all * diagnosis and all waveforms, type 17 between all waveforms and all other diagnosis, and type 18 between * diagnosis and all other diagnoses.

TABLE 5.6 INTRA-TEST PRECEDENCE RELATIONSHIPS

EDGES			SELECTION RULE	
TYPE	PRIOR-ITY	RELATIONSHIP	PREDECESSOR	SUCCESSOR
1	1	Data-determinacy	Source variable in an assertion	The assertion
			An assertion having a target variable	The target variable
			A variable in a parameter of diagnosis	The diagnosis
			A diagnosis	Variable in operator response of the diagnosis
2	2	Waveform-setup	Stimulus-conjunction	Measurement-conjunction
16	5	Waveform-diagnosis	Diagnosis selected by * logic	All waveforms
17	5		All waveforms	All diagnoses not selected by * logic
18	5		Diagnoses selected by * logic	All other diagnoses
19	1	Hierarchical	Node in an input structure declaration	All its direct descendent nodes
			Node in an output structure	Its parent node in the structure

TABLE 5.6 (continued)

20	1	Pointer	Pointer variable (i.e. variable with prefix PTR_ or PTRy_, where y is a digit 1 to 5)	Structure for which the pointer variable is a key or parameter (given by the suffix of the pointer variable)
21	6	Recursive*	A source variable in an assertion with subscript of the form I-k, where k is positive integer	The assertion

* This relationship is originally entered as type 1, but is changed later to type 21 in subscript analysis phase.

ALGORITHM 5.1 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, assertions 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.

Algorithm 5.2 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 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 S2 to S2.3.2). 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.

ALGORITHM 5.2 DETECT AND ENTER WAVEFORM SETUP, DATA DETERMINACY, HIERARCHICHAL, POINTER, AND WAVEFORM-DIAGNOSIS RELATIONSHIPS FOR A TEST MODULE

S0: Define data determinacy relationship as precedence type 1, priority 1, waveform setup relationship as precedence type 2, priority 2, hierarchichal as type 19, priority 1, pointer as type 20, priority 1, waveform-diagnosis as type 16, 17 and 18, priority 5.

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.

ALGORITHM 5.2: (continued)

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: /* Hierarchichal relationships */

For each of the variables found in the earlier steps repeat the following:

S4.1 Let V_i be a variable found in earlier steps. For V_i find all its ancestors by traversing the tree given by the structure declaration and enter each node sequentially in ancestlist. Also create space for a new entry in W if the node is not yet in W.

S4.2 If V_i belongs to an input structure then for each of the ancestors i do W(ancestlist(i) ancestlist (i-1))=19 else for each of the ancestors i do W (ancestlist(i-1), ancestlist(i))=19

S5: /* pointer relationships */

For each of the variables names V_i in W do the following: (Let i be the position of V_i in W)

S5.1 If prefix of the variable name V_i is 'PTR_' then do S5.11

S5.11 Search for the variable with the name same as suffix after 'PTR_' in V_i (or 'PTRx_', where x is a digit).

S5.12:If not found then issue a warning saying that the structure corresponding to the V_i is never used. Go to S5.11

S5.13:Let j be the variable for which match was found.

v(i,j) = 20

```
S6:  /* Waveform - diagnosis relationships */ do for
     all diagnosis and waveform
     S6.1    Let i be the number assigned to a waveform.
             Let j be the number assigned to a diagnosis.
             If j is selected by '*' logic then
                       W(j,i) = 16
                else   W(i,j) = 17
S7:  do for all diagnosis
     S7.1    Let i be a diagnosis selected by *logic.
             Let j be a diagnosis not selected by '*' logic.
                       W(i,j) = 18
S8:  Return
```

5.2.3 Graph Analysis

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 5.5 shows the matrix for the test module FREQ in the sample test specification MINIRADIOSET of Figure 4.7. To the left of the matrix are node numbers with the corresponding 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;

= otherwise.

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

(1) 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 exclusive 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 5.5 WEIGHTED ADJACENCY MATRIX FOR TEST MODULE FREQ.

(2) 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).

(3) 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.

(4) 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 (3) and (4) 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 (3) and (4) need not be checked. If there is more than a TARGET variable, then case (2) must have already detected, hence (3) and (4) 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 will skip the subphases of cycle detection, subscript

analysis, and sequence determination and proceed to the next module.

5.2.4 Cycle Detection and Elimination (This subphase is executed after the subscript analysis phase:5.2.5)

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 cycles and eliminate them automatically, if possible; and 2) report to the user about erroneous circular definitions.

In order to do this task, a path matrix (or reachability matrix) of the digraph is generated. A path matrix, P, is an n by n 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.

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).

An efficient way of generating the path matrix from an adjacency matrix is given by Warshall's algorithm (Algorithm 5.3)

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 S4 denotes logical OR.

Once the path matrix P has been created, any cycle 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 subphase of sequence determination. When there are cycles each distinct cycles 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 there 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. When edges are removed from the graph, the user is informed about the modification through a message (Message #16). Eventually, either the digraph becomes cycle free, or all the unbreakable cycles are detected and reported to the user. The Algorithm 5.4 identifies and enumerates distinct cycles, and tries to break them if possible. The expansion involves the automatic cycle elimination once a cycle is identified (steps S24 to S24.7 of Algorithm 5.4).

ALGORITHM 5.3: WARSHALL'S: CREATE PATH MATRIX

P = Path Matrix

W = Weighted Adjacency Matrix

A = Adjacency Matrix

n = number of nodes in digraph

S1: Let P(i,j) = 1 if W(i,j) > 0
else P(i,j) = 0 (for all i and j). (Note that P(i,j)=A(i,j)

S2: For each i=1 to n do steps could have been used instead).
S3 to S6.

S3: For each j = 1 to n do steps S4 to S5.

S4: If P(j,i) = 1 then set
P(j,k) = P(j,k) | P(i,k)
(for all k=1 to n).

S5: End loop on j.

S6: End loop on i.

S7: Return.

The algorithm has five inputs: (1) the total number of nodes in the digraph, n; (2) the adjacency matrix, A; (3) the path matrix, P; (4) the weighted adjacency matrix W; and (5) the precedence priority vector PRIORITY. The first three parameters are needed in enumerating all distinct cycles; while the last two parameters are required for 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 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) If it finds that the edge is of type 21 it means that it is a recursive edge (i.e.: A(I) = A(I-k) is represented as in Figure 5.6)

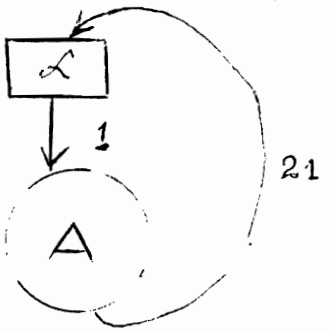


Figure 5.6 (Recursive Edge)

It should be noted here that originally this was entered as 1, but it was marked as 21 by the subscript analysis SUBANAL.). In such a case it stores the cycle in an array CYCLIST, which is used later by the sequencing algorithm. (See Table 5.6 for description of edge of type 21.)

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 5.7 shows a digraph, together with all the trees constructed by the algorithm, the cycles eliminated, and the cycles printed out. All the

ALGORITHM 5.4 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.

ALGORITHM 5.4 (continued)

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 precedence 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 */

ALGORITHM 5.4: (continued)

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 If W(p,s) = 21 then do;
cyc = #cyc + 1
Store the nodes in a cycle into a row of
the array CYCLIST(#cyc,*).
For each of the nodes i in the cycle c,
set INCYC(i) = c.

S24.5 /* Delete the edge by zeroing A(p,s) & W(p,s) */
Set A(p,s) = 0 and W(p,s) = 0.

S24. /* 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. Set LEVEL = d.

S24. Goto step S16.

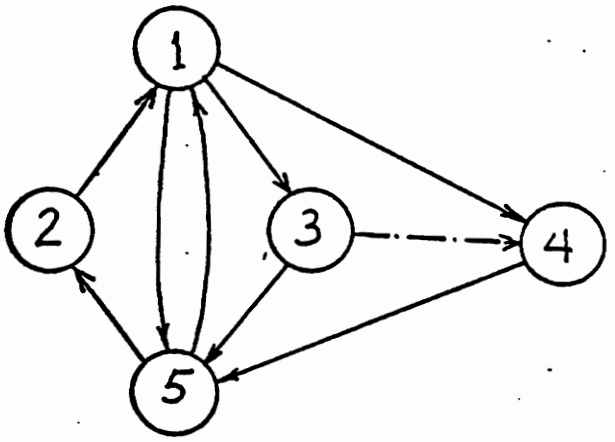
S24. /* All edges have priority 1; print the cycle */
print the cycle: PATH(k), for k=1 to level
(Message #5).

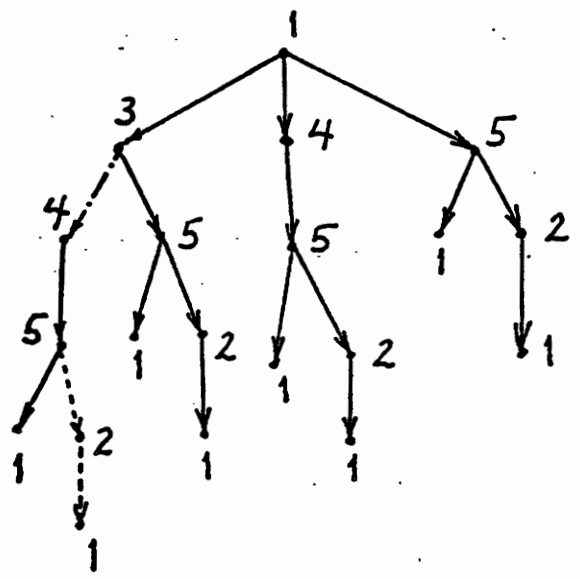
S25: Goto S13.

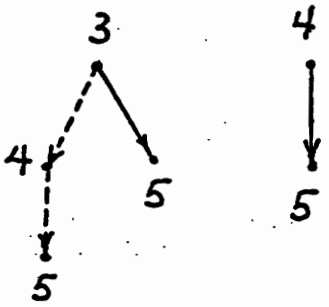
S26: Set ROOT = ROOT +1.

S27: If ROOT < = n then go to S2.

S28: Return.







Edges with priority 1

Edges with priority 2

Additional tree edges that would have been constructed if all edges had priority 1

CYCLES ELIMINATED: 13451, 134521

CYCLES PRINTED: 1351, 13521, 1451, 14521, 151, 1521

Figure 5.7 Cycle Enumeration and Elimination of a Sample Graph

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.

Having created the adjacency matrix, analyzed it for consistency and completeness, and enumerated/eliminated cycles, the Processor finishes the phase of 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 subscript analysis and sequence determination. Otherwise, the Processor cannot proceed until the corrected test specification is submitted. In this case, it should have provided the user with various reports indicating possible causes of the problems and suggestions for corrections.

5.2.5 Processing of Subscripts (This phase is executed before 5.2.4)

This section discusses the syntactic and semantic checks of the statements which use subscripted variables. Prior to this phase, adjacency matrix A(N,N) has been created.

Inputs from previous stages to this program SUBANAL are the following:

1. N --- The size of the adjacent matrix.
2. A(N,N) --- Adjacent Matrix that specifies precedence between nodes.
3. #STIM --- The number of stimuli in this test.

4. #MEAS --- The number of measurements in this test.
5. #DIAG --- The number of diagnoses in this test.
6. #V --- The number of variables (local and global) in this test.
7. HOM_NODE(N) --- Directory location of each node in the adjacent matrix.
8. HOM_PTR(N) --- Storage entry pointer for each STIM/MEAS/DIAG node, and TP(DCL) pointer for each variable.

In addition to the data shown above, SUBANAL also accepts inputs generated by the external sequencing procedure EXTSEQ, and the associative memory produced in syntax analysis phase. Inputs from EXTSEQ include:

1. #GLVAR --- the number of global variables in the test specification.
2. GL_VAR --- the table for all the global variables. GL_VAR has a similar structure to LCL_VAR which is described later.
3. #REDFUN --- the number of reduction functions known to the system.
4. REDFUNC(#REDFUN) --- names of all defined reduction functions.

The input parameter TESTPTR is passed to SUBANAL when it is called. TESTPTR is a pointer to the test under consideration.

Program SUBANAL accepts the above data as inputs and generates some data, e.g. dictionary entry DICT(N) for each node for later use. It also checks the correct usage (both syntactically

and semantically) of subscripted variables. SUBANAL is subdivided into four parts as shown in the Figure 5.8. CRDICT creates some data structure such as local variables table, dictionary entries, etc. Procedure PARSE_SUB parses the subscript declaration statement. Procedure SUBUSAG checks the correct usage of subscripts in the waveforms within the test module. The usage of reduction function is checked in the procedure REDUSAG. These routines are further described in the sequel.

The following data are generated by Algorithm SUBANAL:

1. STIM_PTR(#STIM)--- A list of pointers to storage entries of conjunction/assertion of all stimuli within the test module.
2. MEAS_PTR(#MEAS)--- A list of pointers to storage entries of conjunction/assertion of all measurements within the test module.
3. D_PTR(#D) --- A list of pointers to storage entries of all diagnoses.
4. LCL_VAR(#LCLVAR)---One entry for each local variable. Each entry has the following subfields.
   - 4.1 VAR_LOC - BIN, the directory location of the variable.
   - 4.2 ARRAY - BIT(1), '1'B if this variable is subscripted, '0'B if the variable is a scalar. The following fields are meaningful only if the variable is not a scalar.
   - 4.3 PTR, a pointer to 'SUBLIST' which identifies the bounds for each subscript used in this variable.

```
        CONJUNCTION $M_W0001($M_DISTORT_V):
            (<J19_A, GND>  = SINE_WAVE(VRMS VOLT ,*,0 SEC ))
                    TARGET: VRMS;

        ASSERTION A1($M_DISTORT_V):
            VRMS >= 2.2
                    SOURCE: VRMS;

        ASSERTION A2($M_DISTORT_V):
            VRMS <= 2.8
                    SOURCE: VRMS;

    LOGIC $LOGIC0010(DISTORT_VOLT): *WAIT, *VRMS_PRNT, |~VRMS_FAILED;

      DIAGNOSIS WAIT:
          OPERATOR MESSAGE:
              TYPE=TUNE_MSG,
              TIME= 0.00000E+00SEC,
          RESPONSE=?;

      DIAGNOSIS VRMS_PRNT:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(VRMS, 'VAC'),
              TYPE=TEXT;

      DIAGNOSIS VRMS_FAILED:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=REF_VOLT(AUDIO_10MW),
              TYPE=VRMS_MSG;

TEST FREQ;

    STIMULI DCV(FREQ);

        CONJUNCTION $S_W0001(DCV):
            (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT ));

    MEASUREMENT $M_FREQ(FREQ);

        CONJUNCTION $M_W0001($M_FREQ):
            (<J22, GND>  = SINE_WAVE(V_SIN VOLT ,FREQ HZ ,DELAY_TIME SEC ))
                    TARGET: FREQ, V_SIN
                    SOURCE: DELAY_TIME;

        ASSERTION $M_W0002($M_FREQ):
            IF DELAY_TIME=60 THEN
                FREQ =  5E+06 +- 60
            ELSE
                FREQ =  5E+06 +- 2.5
                    SOURCE: FREQ, DELAY_TIME;

    LOGIC $LOGIC0010(FREQ): *GET_DELAY_TI, |~FREQ_TOL_FAI, *FREQ_PRNT;
```

FIGURE 4.7: (continued)

```
        DIAGNOSIS GET_DELAY_TI:
            OPERATOR MESSAGE:
                TYPE=WARMUP_MSG,
                TIME= 0.00000E+00SEC,
            RESPONSE=(DELAY_TIME);

        DIAGNOSIS FREQ_TOL_FAI:
            OPERATOR MESSAGE:
                AFFECTED COMPONENTS=FREQ_TOL(STD_5MHZ_FRE),
                OTHER PARAMETERS=('FREQ'),
                TYPE=FREQ_TOL_MSG;

        DIAGNOSIS FREQ_PRNT:
            OPERATOR MESSAGE:
                OTHER PARAMETERS=(FREQ, ' HZ'),
                TYPE=TEXT;

    TEST DISTORT_10MW;

       STIMULI $S_DISTORT_1(DISTORT_10MW);

          CONJUNCTION $S_W0001($S_DISTORT_1):
              (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &
              (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

       MEASUREMENT $M_DISTORT_1(DISTORT_10MW);

          CONJUNCTION $M_W0001($M_DISTORT_1):
              (<J19_A, GND>  = DISTORTION(M_DISTORT % ,2 KHZ ))
                      TARGET: M_DISTORT;

          ASSERTION $M_W0002($M_DISTORT_1):
              M_DISTORT <= 3
                      SOURCE: M_DISTORT;

       LOGIC $LOGIC0010(DISTORT_10MW): *DISTORT_PRNT, |~AUDIO_DISTOR;

        /*** FOLLOWING DIAGNOSIS ALREADY DEFINED BEFORE:

          DIAGNOSIS DISTORT_PRNT:
              OPERATOR MESSAGE:
                  OTHER PARAMETERS=(M_DISTORT, '%'),
                  TYPE=TEXT;
                                                              ***/

          DIAGNOSIS AUDIO_DISTOR:
              OPERATOR MESSAGE:
                  AFFECTED COMPONENTS=DISTORT(AUDIO_10MW),
                  OTHER PARAMETERS=(' 10MW', 1.0),
                  TYPE=DISTORT_MSG;

       /*************************************************/
       /*                                               */
       /*  MESSAGES                                     */
```

FIGURE 4.7: (continued)

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


   MESSAGE SHORTED_MSG:
       TEXT='R/T DC INPUT SHORTED J24-B/J24-C ', 'AN/GRC-106 DEFECTIVE.  CHECK P
RINTOUTS FOR DEFECTS.', 'PRESS STOP.';

   MESSAGE TEXT:  ALIAS=DISPLAY,
       TEXT='(P1):   (P2) ';

   MESSAGE FREQ_TOL_MSG:
       TEXT='(C) DEFECTIVE.', '5.0 MHZ STD. OUT OF (P) TOLERANCE.';

   MESSAGE DISTORT_MSG:
       TEXT='(P1) AUDIO DISTORTION GREATER THAN (P2) PERCENT.';

   MESSAGE TUNE_MSG:
       TEXT='TUNE RECEIVER:  MC & KC CONTROLS TO 250000.', 'ADJUST AUDIO GAIN CO
NTROL FOR 2.2 TO 2.8 VAC', '(2.5 VAC NOMINAL).    PRESS  YES.';

   MESSAGE VRMS_MSG:
       TEXT='10 MW DISTORTION REFERENCE VOLTAGE FAILED.';

   MESSAGE WARMUP_MSG:
       TEXT='IF A 12 MINUTE WARMUP IS DESIRED, ENTER 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 00600: AUDIO_2W, FAILURE FUNCTION=DISTORT;

      /******************************************************/
      /*                                                    */
      /*  UUT CONNECTION POINTS                             */
      /*                                                    */
      /******************************************************/
```

FIGURE 4.7: (continued)

```
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      : J16, CONNECTOR=(COAXIAL, ),
        LIMIT=(UVOLT,  1.00000E+02,  0.00000E+00, GND),
        COMMENTS=' COAXIAL CABLE';

UUT_POINT      : J19_L, LIMIT=(VOLT,  7.00000E+01,  0.00000E+00, GND);

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: OHMMETER, FUNCTION TYPE=M, #PINS= 2,
        PARAM_01=(X, T, LIMIT=(OHM,  1.00000E+03,  1.00000E+00));

FUNCTION   120: AMPL_TOL, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    10: PWR_SUPPLY, FUNCTION TYPE=S, #PINS= 2,
        PARAM_01=(X, S, LIMIT=(VOLT,  6.00000E+01,  0.00000E+00));

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    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));

FUNCTION   140: DISTORT, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

FUNCTION    30: SINE_WAVE, 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='AMPL., FREQ., TIME DELYD';

FUNCTION   130: REF_VOLT, FUNCTION TYPE=F,
```

FIGURE 4.7: (continued)

```
            PARAM_01=(COMPONENT, S);

    FUNCTION   110: FREQ_TOL, FUNCTION TYPE=F,
            PARAM_01=(COMPONENT, S);

      /*************************************************/
      /*                                               */
      /*  ATE CONNECTION POINTS                        */
      /*                                               */
      /*************************************************/


    ATE_POINT       : ATE_J24B, UUT_POINTS=(J24_B);

  END MINIRADIOSET;
```

FIGURE 4.7: (continued)

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).

4.5.3 Syntax Error/Warning Report

The Syntax error/warning 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 4.3 this report is generated by a collection of error message routines (see Section 4.3.2). 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 4.4. The warning messages indicate the assumptions made, or the corrective actions taken by the Processor. All messages issued during the different phases are collected in one file.

4.6. Cross Reference Reports

This section presents the subsystem which derives, checks the data types and produces a set of six reference reports as summarized in Table 4.1. As indicated in Figure 4.3, these reports are generated by a program module XREF (XREF1 and XREF2) whose input is the stored NOPAL specification. Section 4.6.1 describes the cross-reference and attribute report produces by XREF1. Section 4.6.2 presents the other cross-reference reports generated by XREF2.

4.6.0 Data Type Derivation and Checking

As well known, the use of data types in programming languages plays an important role in programming documentation and mistake prevention. In the NOPAL system, we used the ideas of "dynamic data type" and "derived data type" and decided to give the user some freedom in choosing the usage of data types.

The basic data types available are restricted by the types available in tho object language EQUETA ATLAS. The user of the NOPAL system can freely

decide what he wants to do about the data types, as long as he uses the variables consistently. That is, he may declare attributes for all the variables in his NOPAL program or declare certain data types for some variables or simply let the system derive all the data types according to the computation specified by the NOPAL assertions.

The basic data types (or attributes of variables) available in the NOPAL system are:

a) INTEGER (abbreviation: INT, code: 4);

b) DECIMAL (abbreviation: DEC, default, code: 1);

c) DIGITAL (abbreviation: DIG, code: 3);

d) BOOLEAN (abbreviation: VOL, 2-valued digital, code: 2).

Note that first, in NOPAL we do not handle the character string operation because the EQUATE ATLAS has no ability in handling this. Secondly, the code 0 is reserved for UNDEFINED DATA TYPE. We describe the implementation of the basic data type checking mechanism in the following.

The data type of a variable can be decided (or derived) by

1. DCL statements, i.e., DCL A digit;
   DCL B integer;...;
2. By the assignment containing the built-in function, i.e., if "A=TRUE" is an assertion then A is of type "dig" and if "I=SUBS('RPES',10);" is an assertion then tye type for I is "INT";
3. By associated operator within the scope of an assertion with most popular mathematical definition for that operator, i.e., if ASSR: A=B+C; then the data type of A, B and C will be of "int" or "dec" depending on the previous use of DCL statement(s);
4. By the assignment containing user-defined functions, i.e., the user has defined a function SINGEN and the "value" clause of the function definition is defined as "DEC", then if A(...)=SINGEN(...) is an assertion the data type of A is assigned as "DEC" as well;

5. By propagation, that is, if "A=B;", "B=C;" and "C=D;" are three assertions and if any one of the three variables has been decided to be of some type then the rest will be decided by propagating the given type.

The check for the use of data types if "operator based" (if we regard the functions as n-ary operators, with n the arity of the function), i.e., the data type of a variable is determined by the use of the variable in assertion. For example, if "A=B*C;" is an assertion, then the data types for B and C are decided by the use of the operator "*", i.e., they must be of type "INT" but if C is declared of type "DIG" or "BOL" which is incompatible with the definition of the operator, an error will be reported and the user is asked to modify the possible misuse of the variable or declaration statement. Finally, the data type for A is propagated from LHS and then of type "INT" or "DEC" depending on the declaration statements for B and C. If none of the variables are declared to be of any data type, the system will assume "reasonable" types for the variables according to the combination of variables and operators. This is why we called the data type check "operator based". Furthermore, the system tries to be as "reasonable" as possible in making the decisions, that is unknown types of variables will be given a type based on the above five criteria for assigning data types; if that effort fails, then "DEC" type will be assumed, because we think that the most generally used type is "DEC". The priority of assigning data types is:

1. Types declared by the user by using DCL statements;
2. Types defined by functions (built-in or user defined);
3. Types implied by corresponding operators;
4. Types propagated by assignment statements;
5. Default.

The table of correspondence between operators, built-in functions and data types is depicted as follows:

LHS(TYPE)	OPERATOR	RHS(TYPE)
"dec" or "int"	+,-,*,/,**	"dec" or "int"
"digtal"	"|,"&	"digital"
"digital"	"+,"*	"integer"
---------	"^	"digital"
"boolean"(s)	|,&	"boolean"(s)
---------	^(not)	"boolean"(s)
"dec" or "int"	<,<=, >,>=	"dec" or "int"
same as rhs	=,^=	same as lhs

BUILT-IN FUNCTIONS: SUBS() → "INT"
TRUE/FALSE → "BOOLEAN".

The implementation:

INPUT: User-specified NOPAL statements (stored in associated memory);

OUTPUT: Array DATATYPES(MSDIRS).

Algorithm:

1) Initialization for DATATYPES;

2) Derive the data types defined by data declaration statements;

3) Derive the data types implied by the use of NOPAL built-in functions;

4) Start the derivation of data types implied by the operators by using two recursive routines for IF-clauses and simple assertions respectively;

5) The collected unknown type variables were entered into TYPESTACK and the propagation of the types among the unknowns is taken place iteratively until there is no further possible derivation;

6) Fill up the rest of the DATATYPES with Code4 (meaning decimal).

Finally, the derived types are passed to XREF1 procedure and printed in cross-reference report.

4.6.1 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 4.8 shows the cross-reference and attribute report from the example given in Figure 4.7.

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 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).

CROSS REFERENCE AND ATTRIBUTES REPORT

NAME	DEF NO.	ATTRIBUTES AND REFERENCES	DATA TYPE
A1	29	ASSERTION LABEL	
A2	30	ASSERTION LABEL	
AMPL	9	TEST LABEL 10 12	
AMPL_TOL	79	ATE-FUNCTION ID , M 14 69	DECIMAL
ATE_J24B	82	ATE-POINT ID	
AUDIO_10MW	70	COMPONENT ID , WITH FAILURE-FUNCTION: REF_VOLT 34	
AUDIO_10MW	71	COMPONENT ID , WITH FAILURE-FUNCTION: DISTORT 52	
AUDIO_2W	72	COMPONENT ID , WITH FAILURE-FUNCTION: DISTORT 23	
CONST_R	74	ATE-FUNCTION ID , M 4	DIGITAL
CONST_S	73	ATE-FUNCTION ID , S 37 26 17 46	DIGITAL
D	53	MESSAGE LABEL 8 13 22 33 44 51	
D2	7	DIAGNOSIS LABEL 6	
D3	8	DIAGNOSIS LABEL	
D4	42	DIAGNOSIS LABEL 41	
D5	43	DIAGNOSIS LABEL 41	
D6	44	DIAGNOSIS LABEL 41	
D7	13	DIAGNOSIS LABEL 12	
D8	14	DIAGNOSIS LABEL 12	
DCV	36	STIMULUS LABEL 37 26	
DCV_AMS	25	STIMULUS LABEL 26 17 46	
DC_INPUT	2	TEST LABEL 3 6	
DISLPAY	53	SYNONYM OF MESSAGE LABEL : D	
DISTORT	81	ATE-FUNCTION ID , M 23 52 71 72	DECIMAL
DISTORTION	76	ATE-FUNCTION ID , M 19 48	INTEGER
DISTORT_10MW	45	TEST LABEL 46 47 50	
DISTORT_2W	15	TEST LABEL 16 18 21	
DISTORT_VOLT	24	TEST LABEL 25 27 31	
F1	39	VARIABLE ID 40 44	DECIMAL

Figure 4.8 First page of cross reference report for MINIRADIOSET.

Any errors or warnings which are detected during this phase of cross-referencing are printed in a separate cross-reference error report. A complete list of the messages are given in Table 4.3.

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. Third, the dimensions of variables occurring in structure declarations are propagated to their decendents.

As described in greater detail in Chapter 5, the variables that are global (or local) to a test module are involved in determining one type of precedence relationship, called data determinancy, in the phases of 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 analysis and sequencing of the test modules is started. If a variable x has ever been defined as TARGET alone, or used as SOURCE alone in any test module, or has a declaration then X is a global variable (i.e. has global scope with respect to the test modules. Otherwise, X is local (i.e. has local scope with respect to each test module where it has been both defined as TARGET and used as SOURCE).

Algorithm 4.4 determines the scope of every variable in the specification. It also designates, as SOURCE, every variable

ALGORITHM 4.4 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 an operator response variable. go to S9.

S6.1 If x is data declaration then
(a) set scope as global
(b) go to S9

S6.2 If x is parameter in test module then
(a) set scope as local
(b) got 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.
```

ALGORITHM 4.4 (continued)

```
S12:  Set 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 S14.1.
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:
        S14.1  If STMT.TYPE is not waveform then go
        to S15.1
S15:  Set     N = 1;
      Set TEST_IDS(1) = the test label of the
          waveform; Go to S17;
        S15.1  If STMT.TYPE is not test or module
               function go to S17
        S15.2  N = 1
               Set TEST_IDS(1) = the label of test
               or module function
               Go to S17;
        S15.3  If STMT TYPE is not data declaration
               then go to S17.
        S15.4  N = 0;/* Not part of any test or module
               function*/
               ND = ND + 1
               DEFN(ND) = TVAL (I)
               REFL (ND) = #REF
               Go to 17
```

ALGORITHM 4.4 (continued)

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

S18: Set TEST_ID = TEST_ IDS(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.

ALGORITHM 4.4 (continued)

```
            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 do S32.
      S32:  /* some residual        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 SO */
S46:    Return;
```

which has not been explicitly declared either as SOURCE or TARGET in the assertions. This relieves the user from explicitly declaring such variable as SOURCE.

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.

4.6.2 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). These 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 4.1). These reports provide better man-machine interface and give the user a clear picture of inter-actions 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. (Figure 4.9)

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. (Figure 4.10).

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

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

Figure 4.9 Diagnosis-Test Cross Reference Report For MINIRADIOSET

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

MESSAGE	DIAGNOSES	TEST-MODULES
#4	D2,	DC_INPUT,
D/DISLPAY	D3, D7, 227, 25, D6, 27,	AMPL, DISTORT_2W, DISTORT_VOLT, FREQ, DISTORT_10MW,
#6	D8, D5,	AMPL, FREQ,
#18	30, 28,	DISTORT_2W, DISTORT_10MW,
#15	24,	DISTORT_VOLT,
#17	26,	DISTORT_VOLT,
#5	D4,	FREQ,

Figure 4.10 Message-Diagnosis-Test Cross Reference Report For MINIRADIOSET

The COMP-DIAG-TEST reports lists all the affected components (i.e., component failures), with all the referencing diagnoses and test modules. (Figure 4.11).

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 (Figure 4.12).

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 4.13).

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 --- 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,
00600: DISTORT(AUDIO_2W)	30,	DISTORT_2W,

Figure 4.9 Affected Component-Diagnosis-Test Cross Reference Report For MINIRADIOSET

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),	

Figure 4.12 UUT Connect Point-Test-ATE Connection PTS
Cross Reference Report For MINIRADIOSET

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),

Figure 4.13 ATE Function Cross Reference Report
For MINIRADIOSET

CHAPTER 5

SPECIFICATION ANALYSIS AND SEQUENCE DETERMINATION

5.1 Overview of Subphases

This phase of the NOPAL processor deals with the analysis of a NOPAL specification and determination of the sequence of events in the program. 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 4.2 will be frequently referred to in the subsequent discussions.

In a NOPAL specification, each entity (e.g. modfun, 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.

The types of precedence relationships dictate the following: 1) how the conjunction and assertions in a test 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 computed in a predecessor test module before it can be used in a successor test module. This kind of precedence information can be represented by a directed graph. There is one separate graph for each one of the test modules and one for the entire collection of test modules in a

modfun. (Note that the present implementation allows one test per modfun in a MODULE, only the MAIN MODULE may have any number of tests in its modfun.) A directed graph consists of nodes and edges. Nodes of this graph represent the entities form the NOPAL specification such as variables, tests, and diagnoses. An edge shows that there is a relationship between a pair of nodes and the label of the edge identifies this relationship. If there is no edge between two nodes it means that they have no immediate relationship.

A precedence relationship indicates that one node must precede the other at execution time of the object test program. Thus one entity is said to be a predecessor of the other, 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 5.1. There are only five relationships within a test module: data determinancy, waveform setup, hierarchichal, pointer, and waveform diagnosis relationships. These are further explained later.

A directed graph (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. The weighted digraph shown in Figure 5.1 corresponds to the inter-test-module relationships of the example of Figure 4.7. In this graph each node represents the name of one of the entities in the NOPAL specification: test modules, diagnoses, and (global) variables. Note that each node may have 0, 1, or more edges emanating from it to successor nodes. The labels of the edges can be interpreted using Table 5.1.

Although a pictorial representation of a graph is convenient for visual study, there are other representations which are better suited for computer processing. The representation called adjacency matrix is used in the analysis phase of the NOPAL processor.

Given a digraph G = < N, E > consisting of a set of n nodes $N = \{V_1, V_2, \ldots, V_n\}$ and a set of edges E, an adjacency matrix, A, corresponding to the digraph G is an n by n matrix such that for all i and j (i,j = 1,2, ... n)

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

$= 0$ otherwise.

TABLE 5.1 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 with global TARGET variable X, (b) Global variable X, (c) Diagnosis with global operator response variable X	(a) Variable X (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 proper subset of D's	D is selected	If D asserts more generic failures, then more specific tests are conducted
6	2		Diagnosis D whose affected components are in conjuction	Test module whose affected components set is a	D is not selected	If D isolates some faults, then skip tests for subset of the same faults
9	3	Stimuli application	Test module which has a stimulus triplet which is globally more frequent	Test module which has a stimulus conjunction with is globally less frequent.		Once a stimuli is applied, as many tests as possible are performed.

TABLE 5.1 (continued)

Precedence			Relationship selection rule		Run-time condition	Explanation
Type	Priority	Strategy name	PREDECESSOR	SUCCESSOR		
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 through 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"(&¬)		

A weighted adjacency matrix 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 integer indicating the type of relationship whenever the corresponding entry in A has a value of 1. A weighted adjacency matrix corresponding to the digraph G is an n by n matrix W such that for all i and j (i,j = 1,2 ... n)

W(i,j) = k iff(V_i, V_j) is an edge of type k in E, i.e. (V_i, V_j) is a relation of type k;

= 0 otherwise

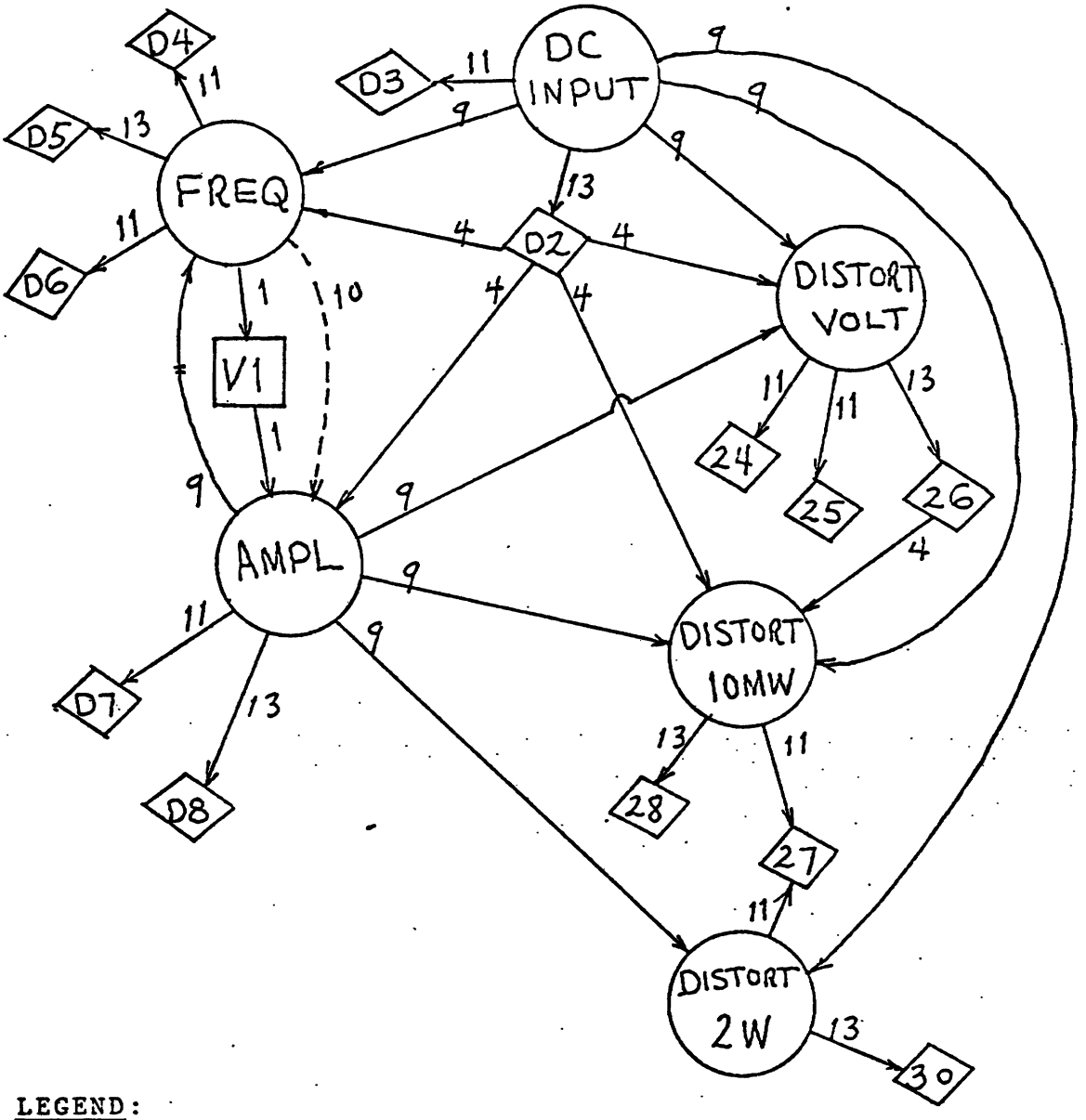


Figure 5.1 Digraph for NOPAL Specification MINIRADIOSET

The weighted adjacency matrix for a NOPAL specification is used extensively in the phases of analysis and sequencing. The matrix corresponding to the MINIRADIOSET example of Figure 4.7 (hence the digraph of Figure 5.1) is shown in Figure 5.2. The node numbers to the left of the node names are assigned by the Processor. Any entry (i,j) in the matrix is either 0, indicating that no relationship exists between node i (at i-th row) and node j (at j-th column), or an integer corresponding to the type of precedence relationships between node i and node j. These type numbers correspond to the precedence types listed in Table 5.1. The precedence types and relationships which are used from the sample specification MINIRADIOSET are enumerated in Table 5.2.

All the global precedence information is conveyed by a weighted adjacency matrix for the whole NOPAL specification. 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.

start insert

As illustrated in Figure ~~3.3~~ 3.6, there are two major subphases of analysis and sequencing: (1) intra-test-module. and (2) inter-test-module. The first subphase deals with only the waveform conjunctions, assertions, and diagnoses of a given test module. ~~In Section 5.2 the subphases of intra-test-module analysis and sequence determination are presented~~. The second subphase deals with the whole collection of test modules in a given NOPAL specification, considering each test module as an integral unit. ~~This phase is needed only for a MAIN MODULE or SPECIFICATION.~~

			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 5.2 WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION MINIRADIOSET

TABLE 5.2 ILLUSTRATION OF PRECEDENCE RELATIONSHIPS FOR MATRIX OF FIGURE 5.2

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.

~~In Section 5.3 the subphases of the inter-test-module analysis and sequencing are presented.~~

A weighted digraph of the NOPAL specification as represented by a weighted adjacency matrix is 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. As these conditions are found, they are printed either as warning or error messages. Further error analysis occurs after the matrix has been constructed. ~~Table 5.3 summarizes the error and warning messages which can be~~ detected 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.

In addition to the above messages, a number of error and warning messages can be issued during subscript analysis. Even though the user may submit a test specification which is correct according to the EBNF syntax of the language, the semantics associated with subscripted variables, test points, and some reserved evaluation functions may indicate erroneous usage. The errors and warning emenating from such cases are detected during the processing of the subscripts in the intra-test-module analysis phase. ~~The context of these messages, and a brief explanation for each case is given in a separate table (Table 5.4)~~

Table 5.5 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 appli-

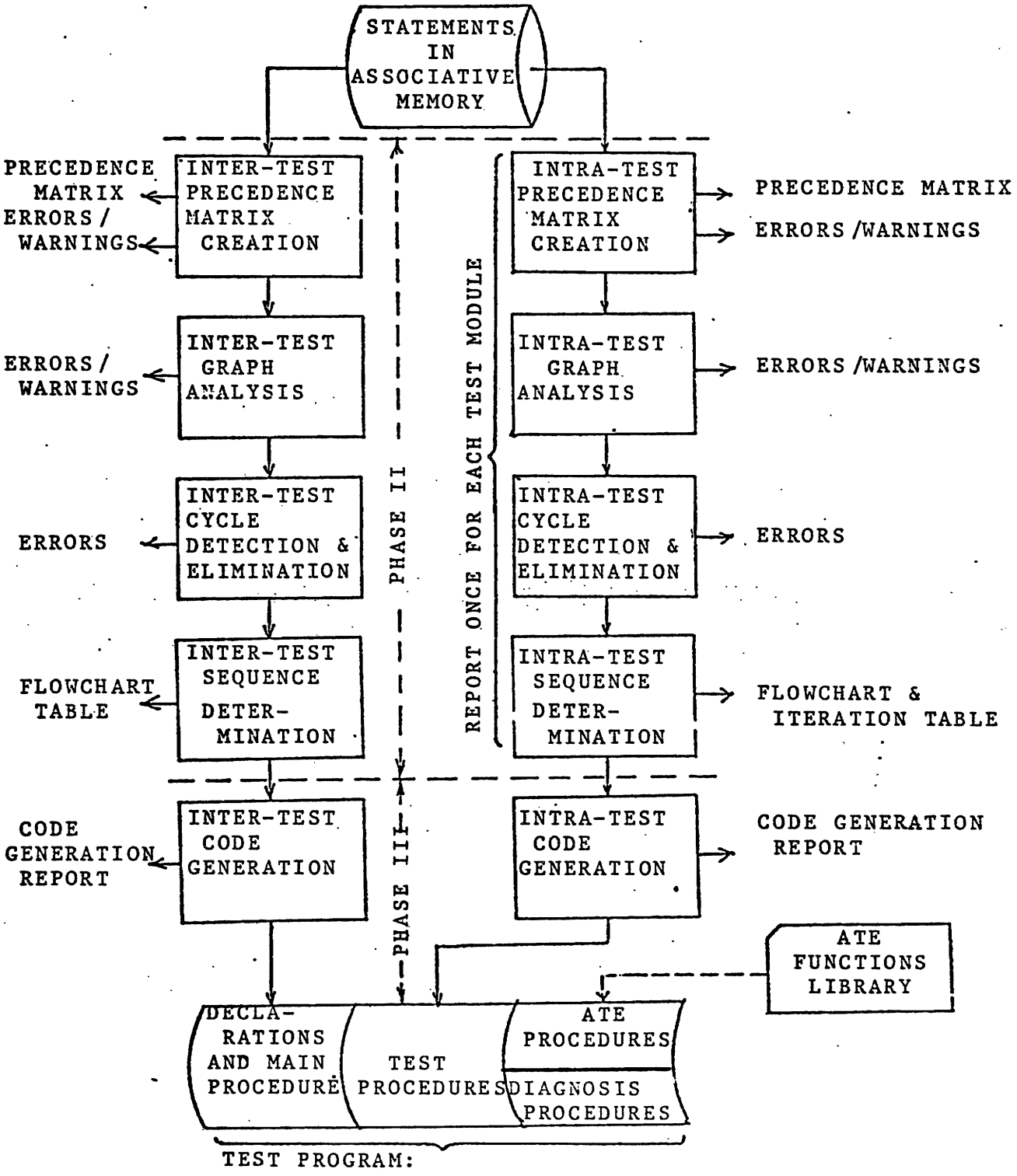


FIGURE 3.6 FLOWCHART FOR PHASES II AND III OF NOPAL PROCESSOR

cable 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 Section 5.2 and 5.3, respectively.

TABLE 5.3

ERROR/WARNING MESSAGES (XREF/ANALYSIS)

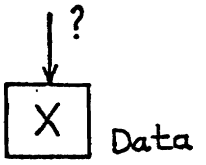
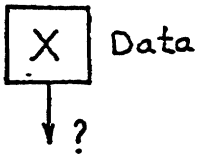
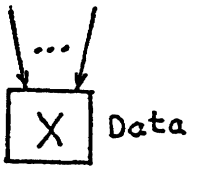
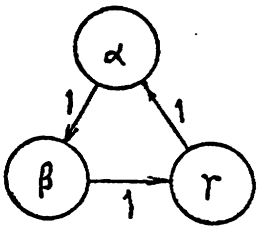
Ref#	Message	Issued by	Brief explanation/Example
1	ERROR (incompleteness): Variable X is used as SOURCE in ...; but its target definition never given elsewhere.	EXTSEQ	
2	WARNING (possible incompleteness): Variable X is defined as TARGET in ...; but never used elsewhere	EXTSEQ	
3	WARNING (possible ambiguity): 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 5.3

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

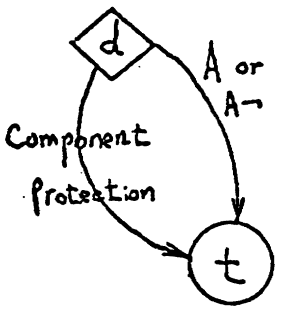
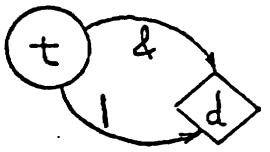
Ref#	Message	Issued by:	Brief explanation/Example
6	ERROR (ambiguity): TARGET variable x in assertion y of test z is <u>not</u> the only expression at the left-hand side of the equal sign ("=")	INTSEQ	W = X + 1 TARGET: X; <u>or</u> X - 1 = W TARGET: X;
7	WARNING (Inconsistency): In assertion x of test 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 5.3

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

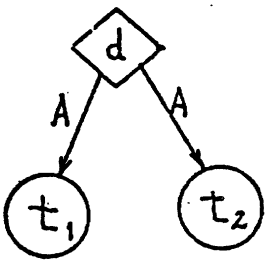
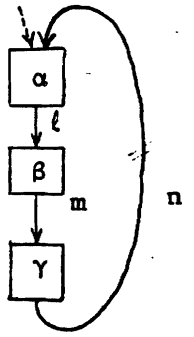
Ref#	Message	Issued by:	Brief explanation/Example
11	WARNING (possible ambiguity): X is defined/used as a local variable in tests...; and also in diagnoses ...	XREF1	Local variable X used in two diagnoses of two different tests.
12	WARNING (apparent inconsistency): In stmnt xxxx, X multiply defined. 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	
16	WARNING (circular definition): The following groups of items α,β,γ... were circularly defined. The cycle was eliminated by deleting item n from the weighted adjacency matrix.	CYCLES	

TABLE 5.4: ERROR AND WARNING MESSAGES FROM SUBSCRIPT ANALYSIS

#(1)

Message:

ERROR(INCONSISTENT):

In test t, the number of dimensions for variable X is inconsistently defined.

When Issued/Examples:

```
Y = A(3,5);
A(I) = X(I)          TARGET:  A(I);
```

The first statement uses 'A' as a two-dimensional array while the second statement uses 'A' as a one dimensional array.

Issued by routine: SUBANAL

#(2)

Message:

WARNING(INCOMPLETE):

Dimension X of variable Y is not declared in test Z, bound undecided.

When Issued Examples:

```
I = SUBS('A:1',10)          TARGET:  I;
K = SUBS('A:3',5)           TARGET:  K;
A(I,J,K)=0                  TARGET:  A(I,J,K);
```

If the second dimension of 'A' has never been declared but is used (e.g. in the form of A(I,J,K)), then this message is issued.

Issued by routine: SUBANAL.

(3)

Message:

ERROR(Ambiguous): In statement number N ---

Subscript declaration is not a simple assertion.

When Issued/Examples:

```
I = SUBS('A:1', 10) + 20              TARGET:  I;
I <= SUBS ('A',5);
```

TABLE 5.4: (continued)

The first statement asserts that 'SUBS' is an arithmetic function that returns a value. The second statement is not an explicit-relational-type assertion.

Issued by routine: PARSE_SUB

(4)

Message:

ERROR(Ambiguous): In Statement Number N ---

Excessive blank appears in the parent list.

When Issued/Examples:

I = SUBS('A B', 10) TARGET: I;

Issued by routine: PARSE_SUB

(5)

Message:

ERROR(Incomplete): In Statement Number N ---

The parent list of subscript declaration is missing or not enclosed in quotes.

When Issued/Examples:

I - SUBS(A,B,10) TARGET: I;

I = SUBS(10) TARGET: I;

The first statement is missing quotes around the parent list. The second statement is missing the parent list.

Issued by routine: PARSE_SUB

(6)

Message:

ERROR(Ambiguous): In Statement Number N ---

Improper use of comma in the parent list.

TABLE 5.4 (continued)

When Issued/Examples

I = SUBS(',A:1',10) TARGET: I;

J = SUBS('A,,B:1' 5) TARGET: J;

Issued by routine: PARSE_SUB

(7)

Message:

ERROR(Incomplete): In statement number N ...

Null parent list is used.

When Issued/Examples:

I = SUBS ('',10) TARGET: I;

Issued by routine: PARSE_SUB

(8)

Message:

ERROR(Ambiguous): In statement number N ...

The dimension number is not a positive integer.

When Issued/Examples

I = SUBS('A:5H',10) TARGET: I;

J = SUBS('A:N',5) TARGET: J;

Issued by routine: PARSE_SUB

(9)

Message:

ERROR(Ambiguous): In statement number N ...

Improper use of colon.

When Issued/Examples:

I = SUBS('A: :2', 10) TARGET: I;

Issued by routine: PARSE_SUB.

TABLE 5.4 (continued)

(10)

Message:

ERROR(Inconsistent): In statement number N ...

The upper bound is not a positive integer or *.

When Issued/Examples

```
I = SUBS('A',B)                    TARGET: I;
J = SUBS('A',-4)                   TARGET: J;
```

Issued by routine: PARSE_SUB

(11)

Message:

ERROR(Inconsistent): In statement number N ...

Variable X is not used as a subscripted variable in the test.

When Issued/Examples:

```
I = SUBS('A',10)          TARGET: I;
A = 0                     TARGET: A;
```

First statement declares A as a subscripted variable. However, A is used as a scalar in the test as the second assertion indicates.

Issued by routine: PARSE_SUB

(12)

Message:

ERROR(Inconsistent): In statement number N ...

Dimension X of variable Y is multiply defined.

When Issued/Examples:

```
I = SUBS('A:2', 10)       TARGET: I;
J = SUBS('B,A:2',5)       TARGET: J;
```

Issued by routine: PARSE_SUB

TABLE 5.4 (continued)

#(13)

Message:

WARNING(Ambiguous): In statement number N ...

Parent name X has never appeared in the test module.

When Issued/Examples:

I = SUBS('A',10) TARGET: I;

When A is declared as a subscripted variable but is never referenced in the test module.

Issued by routine: PARSE_SUB

#(14)

Message:

ERROR(Incomplete): In statement number N ...

Subscript declaration is not correct, check use.

When Issued/Examples:

I = SUBS('A') TARGET: I;

When the subscript is improperly declared in addition to the previous cases (i.e. messages 3 to 13), this general message is issued. In the example, the declaration is incomplete because the bound field is missing.

Issued by routine: PARSE_SUB

#(15)

Message:

ERROR(Ambiguous): in statement number ---

Variable X is multiply declared as free subscript.

When Issued/Examples:

When a variable is declared more than once within the same test module, then we issue the above message.

e.g.

I = SUBS('A:1', 10) TARGET: I;

TABLE 5.4 (continued)

I = SUBS('A:2', 5) TARGET: I;

Issued by routine: PARSE_SUB

(16)

Message:

ERROR(Inconsistent): in statement number N ...

The free subscript X in source variable does not appear as free subscript in target variables.

When Issued/Examples:

A = B(I) TARGET: A SOURCE: B(I), I;

When I is declared as a free subscript, the above assertion will repeat many times. Then an error occurs since A is multiply assigned a value.

Issued by routine: SUBUSAG.

(17)

Message:

ERROR(Inconsistent): in statement number N ...

Subscript in variable X does not follow the correct syntax.

When Issued/Examples:

C = D(3.5) TARGET: C SOURCE: D(3.5);

The assertion uses a non-integer '3.5' as subscript. Thus the above error message is issued.

Issued by routine: SUBUSAG

(18)

Message:

WARNING(Possible inconsistent): in statement number N ---

I-th subscript of variable X is a subscripted variable or a non-free subscript; range test is not performed.

TABLE 5.4 (continued)

When Issued/Examples:

A(I) = B(C(I)) TARGET: A(I) SOURCE: B(C(I)), C(I), I;

A(I) < B(X) SOURCE: A(I), B(X), I, X;

Suppose B is declared as a vector of size 10. Since we do not know the value of C(I) and X until run time, range test cannot be performed at compile time.

Issued by routine: SUBUSAG.

(19)

Message:

ERROR(Inconsistent): in statement number N ---

Variable X is not found in the variable table or dimension-I is not defined.

When Issued/Examples:

This message is isssued when a variable is not declared as being subscripted (through subscript declaration statement) and is used. For example, if A is not declared as a vector, then

A(5) > 0 SOURCE: A(5)

Causes an error.

Issued by routine: SUBUSAG.

(20)

Message:

ERROR(Inconsistent): in statement number N ---

The subscript bound of variable X for dimension-I was declared to be M, but N is used here.

When Issued/Examples:

When the actual subscript used is greater than the upper bound declared, the above message is issued.

TABLE 5.4 (continued)

I = SUBS('A:1', 5) TARGET: I;

A(10) > 5 SOURCE: A(10);

Issued by routine: SUBUSAG

#(21)

Message:

ERROR(Ambiguous): in statement number N ... .

Reduction function is not used in a simple explicit assertion.

When Issued/Examples

X < SUM(A(I),I) SOURCE: X,A(I),I;

IF VAR > 60 THEN Y = SUM(A(I),I) ELSE Y=SUM(B(I),I) TARGET: Y;

The first statement is not an explicit relation assertion. The second statement is a conditional assertion.

Issued by routine: REDUSAG.

(22)

Message:

ERROR(Ambiguous): in statement number N ...

Instead of 2, P arguments are used in the reduction function.

When Issued/Examples:

X = SUM(A(I), B(I), I) TARGET: X;

Y = SUM(C(I,J)) TARGET: Y;

The reduction function (e.g. SUM) should always have two arguments. The first statement has three arguments while the second statement has one.

Issued by routine: REDUSAG

(23)

Message:

ERROR(Ambiguous): in statement number N ...

The second argument X of the reduction function is not declared as a free subscript.

TABLE 5.4 (continued)

When Issued/Examples:

```
X = SUM(A(I), B(I))                    TARGET:  X;
Y = SUM(A(I),10)                       TARGET:  Y;
```

Issued by routine: REDUSAG

(24)

Message:

ERROR(Ambiguous): in statement number N ...

The first argument X of the reduction function is not a subscripted variable.

When Issued/Examples:

```
X = SUM(A,I)                           TARGET:  X;
Y = SUM((SUM(I,J),J),I)                TARGET:  Y;
```

First statement has 'A' a simple variable, as the first argument of a reduction function. Second statement uses a function as the first argument.

Issued by routine: REDUSAG

(25)

Message:

ERROR(Ambiguous): in statement number N ...

The I-th subscript X of variable Y is not a simple variable or constant.

When Issued/Examples:

```
X = SUM(A(C(I)),I)                     TARGET:  X;
Y = SUM(A(I,J+2),I)                    TARGET:  Y;
```

The first statement uses C(I) as subscript while the second statement uses J+2 as subscript.

Issued by routine: REDUSAG

TABLE 5.4 (continued)

#(26)

Message:

ERROR(Ambiguous): in statement number N ...

More than one dimension may be reduced in a reduction function.

When Issued/Examples:

X=SUM(A(I,J,I),I) TARGET: I;

'I' is the first and third subscripts of variable 'A'; we do not know which dimension is to be reduced.

Issued by routine: REDUSAG

#(27)

Message:

ERROR(Ambiguous): in statement number N ...

The subscript being reduced X is not found in the first argument Y.

When Issued/Examples:

X=SUM(A(I),J) TARGET: X;

Y=SUM(A(I,J),K) TARGET: Y;

This message is issued when the second argument does not appear in the subscript list of first argument; we do not know which dimension is to be reduced.

Issued by routine: REDUSAG

#(28)

Message:

ERROR(Inconsistent): in statement number N ...

The dimensionality of LHS is not one less than that of RHS.

When Issued/Examples:

X(I) = SUM(A(I,J,K),J) TARGET: X(I);

Y = SUM(A(I,J),I) TARGET: Y;

Issued by routine: REDUSAG.

TABLE 5.4 (continued)

#(29)

Message:

ERROR(Ambiguous): in statement number N ...

The LHS X of a reduction function is not a subscripted variable or scalar.

When Issued/Examples:

SIN(X) = SUM(A(I),I)

When LHS of an assertion is not a (subscripted) variable, this message is issued.

Issued by routine: REDUSAG.

#(30)

Message:

ERROR(Inconsistent): in statement number N ...

Subscript X in left-hand side of a reduction function does not appear in right-hand side.

When Issued/Examples:

A(I) = SUM(B(I,J),I) TARGET: A(I);

Subscript I is reduced and should disappear from subscript list of LHS. Instead, A(J) should have been the target variable.

Issued by routine: REDUSAG

#(31)

Message:

ERROR(Ambiguous): In statement number N ...

Reduction function is improperly used.

When Issued/Examples:

When the usage of a reduction function is incorrect in addition to the reasons cited before (messages 21 to 30), this general message is issued.

TABLE 5.4 (continued)

X = SUM(A(I),I)+5 TARGET: X;

Y = 2*SUM(A(I),I) TARGET: Y;

Both of the above examples include an arithmetic operator in RHS of the assertion.

Issued by routine: REDUSAG.

#(32)

Message:

ERROR(Incomplete): in test X

Not all nodes are rankable, return from PRECED.

When Issued/Examples:

When some nodes are not rankable due to circular definition or incorrect usage of subscripts, this message is output.

Issued by routine: SCHEDLER

#(33)

Message:

ERROR(Inconsistent):

Global variable V is used as an array, but with different number of dimensions. Initially used with DIM# (i) number of dimensions.

When Issued/Examples:

A(I,J) < 5 SOURCE: A(I,J);

A(I) = VAR TARGET: A(I);

Global variable A is used as both 2-dimensional and 1-dimensional array: then this message is issued.

Issued by routine: EXTSEQ

TABLE 5.4 (continued)

#(34)

Message:

ERROR(Inconsistent):

Global variable V is used as an array and scalar, it cannot be both.

When Issued/Examples:

```
A = 5            TARGET: A;
A(I) > 6         SOURCE:  A(I);
```

Global variable A is used as both a scalar and an array. Then this message is issued.

Issued by routine: EXTSEQ

TABLE 5.5 SUMMARY OF 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 an n by n 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.	Subscript Analysis	Check the syntax and semantics of the usage of subscripted variables.
5.	Cycle Detection and Elimination	Create path matrix from A; search for possible cycles; delete the cycles if possible; otherwise report as error to user.
6	Sequence Determination	Rank the nodes of the digraph according to precedence, and then reorder the nodes by their rank.

5.2 Intra-Test-Module Analysis and Sequencing

5.2.1 Overview of Intra-Test-Module Analysis

This section provides in greater detail the subphases of inter-test-module analysis. These subphases as shown in Figure 5.4 include the graph creation, analysis, subscript processing, and sequencing. The overall program which performs intra-test-module analysis and sequencing is named INTSEQ . INTSEQ generates, as output, a vector, ORDER , of nodes ordered in their execution sequence and an iteration table, DOTAB , which identifies the iteration variables and iteration scopes. INTSEQ also generates messages if errors are detected in each subphase. The error and warning messages issued by the first four lines of Figure 5.4 are referred to by the reference numbers given in Table 5.3, while the messages issued by subscript analysis, subscript propagation, and sequencing are listed in Table 5.4.

Section 5.2.2 discusses the creation of the graph (in adjacency matrix form) and the entering of precedence relationships. Section 5.2.3 deals with preliminary analysis of the graph. Section 5.2.4 discusses the subjects of cycle detection, enumeration, and elimination. The processing of subscript statements is described in detail in Section 5.2.5. Section 5.2.6 presents the method of subscript propagation, and section 5.2.7 discusses the rationale and methodology of determining execution sequence. An example of a flowchart report as well as a brief explanation of the report are provided in section 5.2.8.

Note that each subphase in INTSEQ must be repeated once for each test module in the whole NOPAL test specification.

Since the current implementation allows only one test per modfun in a MODULE, sequencing this test implies sequencing the modfun.

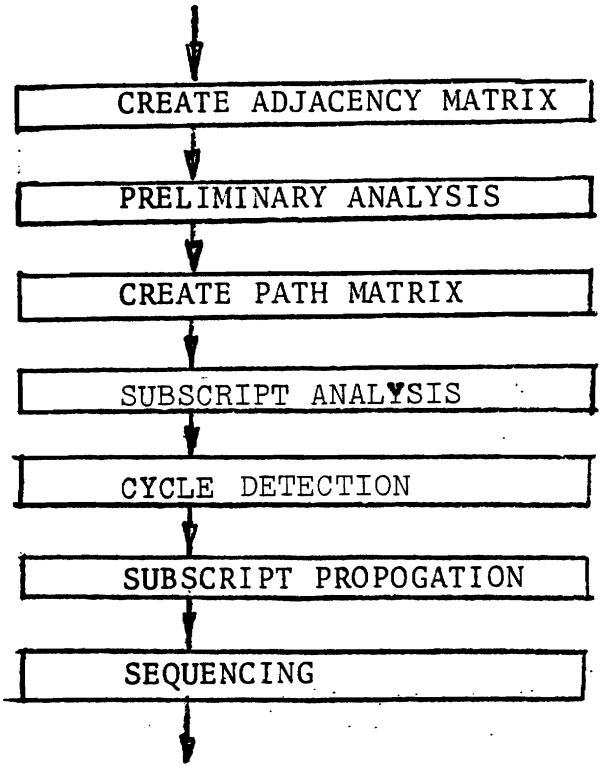


FIGURE 5.4 FLOWCHART OF INTRA-TEST MODULE ANALYSIS AND SEQUENCING

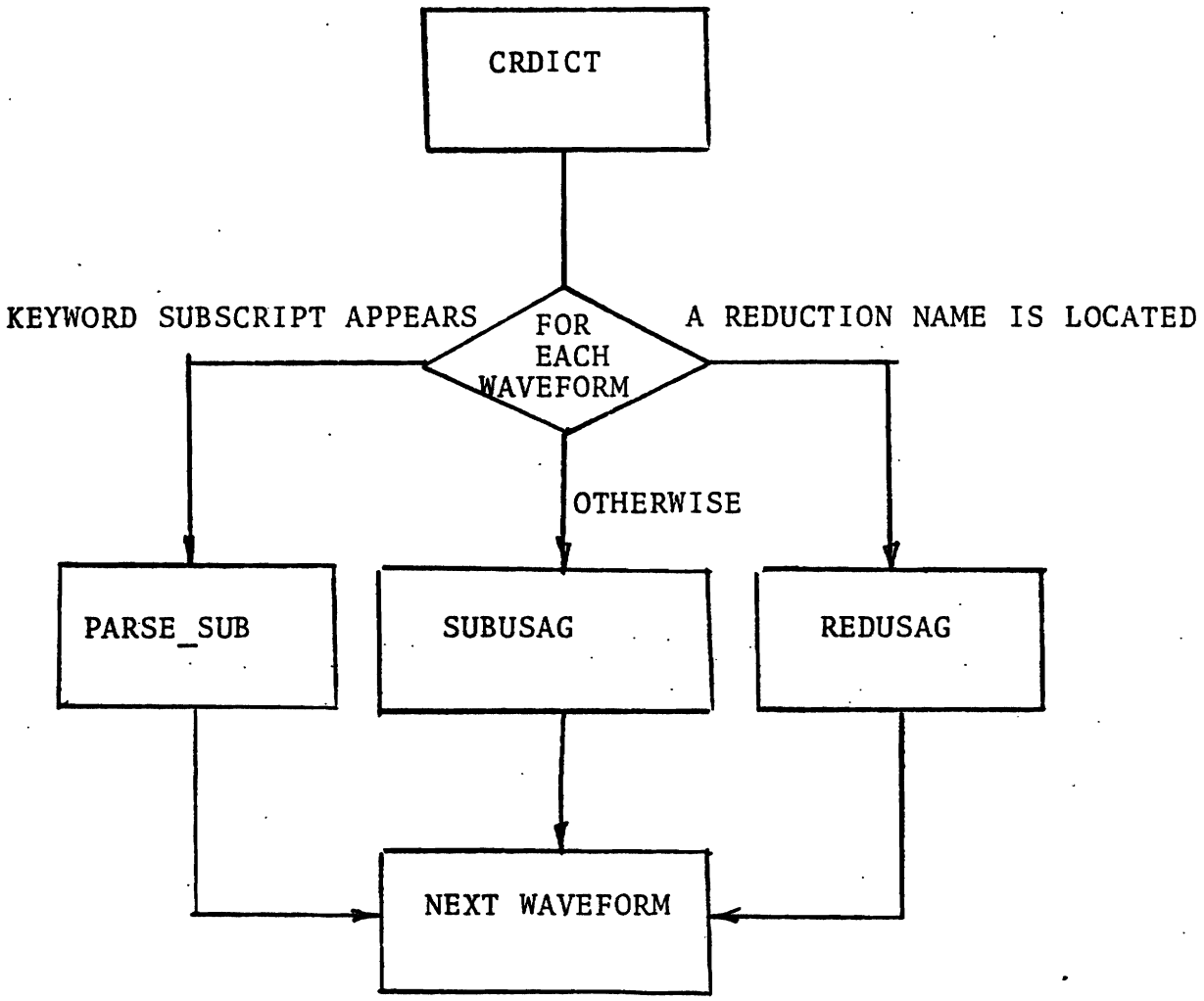


FIGURE 5.8 FLOWCHART OF SUBSCRIPT ANALYSIS PROGRAM-SUBANAL

4.4 EC_FLD - BIT(1) - '1'B if the variable name serves as a control/evaluation function.

4.5 D_TYPE - CHAR(1) - the attribute of this variable. For example, 'I' means integer, 'C' means character, and 'B' means bit etc.

5. NFOR --- BIN, the number of free subscripts declared in this test module.

6. FSUB(NFOR) --- There is one entry for each free subscript. Each entry has the following subfields.

6.1 IXDIR - BIN, directory location of the subscript name.

6.2 LFORECH - CHAR, the name of the subscript

6.3 UPBOUND - BIN, upper bound of the subscript.

7. DICT(n) --- A dictionary entry for each node in the adjacent matrix. It has the following subfields:

7.1 DNAME - BIN, the directory entry of this node.

7.2 DTYPE - CHAR(1). It indicates the type of the node. For example, 'C': stimulus conunction; 'A': stimulus assertion; 'X': measurement conjunction; 'Y': measurement assertion; 'D': diagnosis; 'L': local variable; 'G': global variable.

7.3 DPTR - PTR, points to storage entry of conjunction or assertion if DTYPE = 'C' or 'A' or 'X' or 'Y'; points to storage entry of diagnosis if DTYPE = 'D'. It points to 'DCL' entry if DTYPE = 'L' or 'G'.

7.4 DVAR - BIN, if the node is a variable, then DVAR is an index to LCL_VAR if DTYPE = 'L' or an index to GL_VAR if DTYPE = 'G'.

As shown in Figure 5.7, Algorithm SUBANAL is the main routine of the subscript processing phase. Procedure CRDICT is imbedded in SUBANAL. SUBANAL first creates dictionary entries for all stimuli, measurements, diagnoses, and variables. If a local variable is encountered for the first time, the procedure SUBANAL creates an entry in local variable list, LCL_VAR. SUBANAL searches the reserved function names 'SUBS' or 'SUBSCRIPT' in each waveform and, if found, calls routine PARSE_SUB to parse the subscripted declaration statement. SUBANAL also searches for reduction function names in assertions. If found, the routine REDUSAG is called to check for the correct usage of a reduction function assertion. For each waveform in the test, SUBANAL calls routine SUBUSAG to check for the consistent usage of subscripted variables. Algorithm 5.5 shows how SUBANAL is implemented.

Routine PARSE_SUB checks the syntax of a subscript declaration assertion. Note that the syntax of subscript declaration should follow the following form:

ITERVAR = SUB ('PARENT', BOUND) TARGET: ITERVAR;

where ITERVAR is the name of free subscript being declared. SUB IS the keyword 'SUBS' or 'SUBSCRIPT'.

PARENT is a list of array variables that use ITERVAR as subscript, and BOUND is the upper bound of the free subscript. Program PARSE_SUB is entered only when keyword 'SUBS' or SUBSCRIPT is detected. The input to this routine is a pointer STO_PTR

ALGORITHM 5.5 SUBSCRIPT ANALYSIS-SUBANAL

FUNCTION: Create dictionary entries and analyze subscript usage.

CALLS: PARSE_SUB, SUBUSAG, REDUSAG.

CALLED BY: INTSEQ.

Step 1. Allocate LOC_TEMP and Initialize its subfields.

Step 2. For each stimulus waveform, fill DICT and STIM_PTR lists.

Step 3. For each measurement waveform, fill DICT and MEAS_PTR lists.

Step 4. For each diagnosis, fill DICT and D_PTR lists.

Step 5. Fill DICT entries for all variables (both local and global).

Step 6. If the variable is declared 'global' then link the variable index in GL_VAR to DICT. Goto Step 11.

Step 7. Now we are dealing with a local variable.
If this variable already has a entry in LOC_TEMP, then go to Step 9.

Step 8. Since this is the first time we encounter this local variable, we create an entry in LOC_TEMP for this variable. We also create SUBLIST for its dimension bounds if it is an array variable.
Goto Step 10.

Step 9. Check whether the number of dimensions is consistent.
If so, go to step 10.
If not consistent, then output error message #1 and return.

ALGORITHM 5.5 (continued)

Step 10. Link the variable index in LOC_TEMP to DICT.

Step 11. For each waveform in stimuli and measurements, search keywords 'SUBS'or 'SUBSCRIPTS' on the right-hand side of an assertion. If found, then call PARSE_SUB to parse the subscript declaration statement.

Step 12. Check whether any dimension in a subscripted variable is undefined. If so, output error message #2 and return.

Step 13. For each waveform in the test, call SUBUSAG to check the consistent usage of subscripted variables.

Step 14. For any function appeared, check whether it is a reduction function. If yes (match through a reduction function table), then call REDUSAG to check its syntax and semantics.

Step 15. Copy LOC_TEMP to LCL_VAR and exit.

which points to storage entry of the waveform under consideration. Upon exit from PARSE_SUB, STO_PTR is unchanged. If the declaration does not follow the correct syntax, then an appropriate error message will be issued. Otherwise, the subscript table FSUB is updated. PARSE_SUB also updates the weighted adjacency matrix W to reflect the dependency of each of the PARENTs to ITERVAR and the dependency of each diagnosis which depends on PARENTs to ITERVAR. Routine PARSE_SUB is shown in Algorithm 5.6.

Routine SUBUSAG checks the usage of subscripted variables. This routine performs the following tests:

(T1) All target variables of a waveform must have the same free subscripts.

(T2) The free subscripts of source variables (either as subscript or as variable) must appear in target variables.

(T3) Subscripts are identified to be in one of the following forms:

--- a subscript term, e.g., I in A(I) or (I-k) where K is a positive integer.

--- an integer, e.g., 5 in A(I,5)

--- a simple variable, e.g., x in A(x) where x is not declared as a free subscript.

--- a generalized expression.

--- another subscripted variable, e.g., B(I) in A(B(I)).

(T4) Verify range if a subscript appears to be a subscript term or an integer.

SUBUSAG also fills the internal attributes S_OR_T and SCOPE in EXPRS for all variables in a waveform. This information will be used later in the code generation phase. SUBUSAGE also changes the entry in the weighted adjacency matrix ω from 1 to 21 if the entry corresponding to it has dependence on (I-K) subscript.

The name of the nodes between which the entry was changed, together with the name of subscript is also entered in a structure RECURCYC. This is used later in code generation.

ALGORITHM 5.6 PARSE SUBSCRIPT DECLARATION ASSERTION - PARSE_SUB

FUNCTION: Checks the syntax of a subscript declaration assertion.

CALLS: None

CALLED BY: SUBANAL

Step 1. Allocate local variables and initialize them

Step 2. If the waveform (pointed by STO_PTR) is not a simple assertion, then issue message #3 and return.

Step 3. Get the subscript name in ITERVAR (LHS of the statement). If it has been declared previously in the same test, then output message #15 and return.

Step 4. Scan the right-hand side of the waveform. Bypass keyword SUBS[CRIPT] and look for parent list which is encloded in quotes. If the quotes are not found, then issue message #5 and return.

Step 5. Check the syntax of parent list. Store parent name and dimension number in PARENTs and PARDIM respectively. Had any error occurred, then an appropriate message is selected and issued from the list #4, #6, #7, #8, #13, or #14, and return. Update the weighted adjacency matrix W. For each PARENT, W(ITERVAR,PARENT) = 1. For each diagnosis, if W(PARENT,diagnosis) = 1 then W(ITERVAR,diagnosis) = 1.

Step 6. Get the upper bound and store it in UPBOUND of the newly created FSUB. If the upper bound is not a positive integer, then issue message #10 and return.

Step 7. For each parent name P in the PARENTs list, search through the dictionary to identify whether it is a local or global variable. Go to corresponding LCL_VAR or GL_VAR table.

ALGORITHM 5.6 (continued)

Step 8. If that variable (P) is not subscripted, then issue message #11 and return. Get to SUBLIST if the variable is subscripted.

Step 9. If the bound of the specified dimension is not defined, then set it to UPBOUND. Otherwise, it is doubly defined and message #12 is issued.

TOPX returns the free subscripts in source and target variables in the lists SFS and TFS respectively. TOPX makes sure that all target variables have the same subscripts. Otherwise, error message #16 is issued. When TOPX is returned to its calling routine, SFS and TFS contains all free subscripts in source and target variables. Then test T2 (i.e. TFS should be a subset of SFS) is performed.

Routine REDUSAG checks the syntax of a reduction function when a reduction function name is recognized in a statement. It takes an input of type 'pointer' which points to the statement (waveform) under consideration. It generates proper error/warning messages as listed in Table 5.4. In addition, REDUSAG produces a matrix REDUCT as its output. REDUCT is an N by NFOR bit matrix. REDUCT(I,J) = '1'B means that J-th subscript is reduced at node I where J is an index to subscript table FSUB and I is an index to dictionary 'DICT'. Algorithm 5.7 depicts the REDUSAG routine.

5.2.6 Subscript Propagation

Program PROPAGT determines the relation between nodes and subscripts in order to find the proper scopes for each possible iteration. This program will construct for each node a list of the subscripts on which it depends, and hence the list of iterations in which it should participate. In the NOPAL language, iterations result from explicit appearance of subscripted variables and subscripts themselves.

The inputs to program PROPAGT include the adjacency matrix A(n,n), subscript list FSUB(NFOR), reduction table REDUCT(N,NFOR) and dictionary entries DICT(N) as described before. The main output is an array called FORCHST(N,NFOR). This array contains

ALGORITHM 5.7: CHECK REDUCTION FUNCTION-REDUSAG

FUNCTION: Checks the syntax of usage of reduction functions.

CALLS: FRES.

CALLED BY: SUBANAL.

Step 1. Initialization and get the waveform pointer in TP.

Step 2. If the waveform is not a simple assertion, then issue message #21 and exit.

Step 3. Let SP point to right-hand side of a reduction function. Now SP should point to a reduction function name. If not, message #30 is issued and exit.

Step 4. The reduction function should have two arguments. If not, a message #22 is issued and exit.

Step 5. The subscript to be reduced is the second argument of a reduction function and its name is put into V2NAME.

Step 6. If V2NAME is not declared as a free subscript in the test modules, then issue message #23 and return.

Step 7. Get to the first argument of the reduction function. If it is not a subscripted variable, then issue a message #24, and return.

Step 8. For each subscript term SNAME of the subscripted variavle, perform steps 9 to 11.

Step 9. If SNAME is not a variable or constant, then issue message #25 and exit.

ALGORITHM 5.7 (continued)

Step 10. If SNAME matches V2NAME, then indicate the dimension number to be reduced and set flag FOUND = '1'B.
Otherwise, test whether SNAME is a free subscript. If yes, then put SNAME onto a list SUBPAT and increment the length L of this list.

Step 11. For next subscript term, goto step 8.

Step 12. FOUND = '0' means that the subscript to be reduced does not appear as a subscript term, message #27 is output and exit is made.

Step 13. So far, the syntax of right-hand-side of the assertion is correct. The free subscript to be reduced is named V2NAME. All other free subscripts are kept in list SUBPAT whose size is L. Now we get to left-hand-side of the assertion.

Step 14. Initialize M to 0 where M is the number of free subscripts in left-hand-side.

Step 15. If LHS is not a variable or an array, then issue message #29 and return.

Step 16. If L=0 and LHS is a scalar, then go to step 21.

Step 17. For each subscript term SNAME of the LHS, perform steps 18 to 20.

Step 18. If SNAME is not a variable or constant, then issue message #25 and exit.

Step 19. If SNAME is a free subscript, then match SNAME in the list SUBPAT. If not matched, then output message #30 and return. If matched, then increment M.

ALGORITHM 5.7 (continued)

Step 20. For next subscript term in LHS, go back to Step 17.

Step 21. For the waveform under consideration, identify its node ID 'AID' into DICT table. The subscript V2NAME to be reduced is also located into offset 'SID' of FSUB table. Set REDUCT(AID,SID) = '1'B.

Step 22. Return.

a positive entry at position (I,J) if the I-th node depends on J-th subscript. I is an index to DICT and J is an index to FSUB. In addition to FORCHST, program PROPAGT also generates the following outputs that are to be used in the later sequencing phase.

(1) PAR_CNT(N) - the count of predecessors of each node.

(2) #ENT - the number of non-zero entries in the adjacent matrix.

(3) SUCESOR(#ENT) - A compact representation of the graph which holds, for each node, a list of its successors. All these lists are placed contiguously in a single list SUCESOR in order to save space. Two auxiliary arrays PLACE1(n) and PLACE2(n) are created to maintain a start index and an end index for each node.

Thus the list of successors of node I contain:
SUCESOR(J), J = PLACE1(I) to PLACE2(I).

(4) PLACE1(N),PLACE2(N) - start and end index in SUCESOR for each node.

(5) INITLST(N) - the initial list used later in the sequencing phase.

PROPAGT is described in Algorithm 5.8.

5.2.7 Sequence Determination

Once the precedence relationships among nodes are established, we are ready to sort them into execution order. A simple topological sorting routine is not adequate. When subscripted variables are involved, the situation becomes more complicated. Procedure SCHEDLR is the main program for performing the sequencing and iteration analysis. SCHEDLR accepts as input the adjacent matrix as well as other data structures generated in the PROPAGT

ALGORITHM 5.8: SUBSCRIPT PROPAGATION - PROPAGT

FUNCTION: Propagate subscripts

CALLS: None

CALLED BY: INTSEQ

Step 1. Calculate the number of non-zero entries in the adjacent matrix.

Step 2. Allocate and calculate the parent count PAR_CNT for each node. Create successors list SUCESOR and its index lists PLACE1 and PLACE2.

Step 3. Compute SUBIND, a mapping of sucscript in FSUB to DICT.SUBIND(I) = J means subscript of the I-th position DICT is the same as J-th position in FSUB.

Step 4. Establish relation between nodes and subscripts. If PRIORITY(subscript,node) = 1 then set FORCHST(node,subscript) = 1.

procedure. SCHEDLR then sorts the nodes into a possible execution sequence, taking into consideration the iteration scopes. As its output, it will produce a vector ORDER of nodes sorted in their ranking order and the loop table DOTAB. Program SCHEDLR depends on the program ORDERER to do the trial and actual ordering, and PRIOANAL to evaluate the results of trial ordering.

5.2.7.1 Procedure - SCHEDLR

The program SCHEDLR initializes variables, and then goes through 3 phases, described below.

Inputs to SCHEDLR

(1) N -- number of nodes to be scheduled.

(2) NFOR -- number of subscripts identified by PROPAGT.

(3) A(N,N) -- adjacency matrix

(4) FSUB(N) -- list of subscripts.

(5) FORCHST(N,NFOR) -- matrix giving dependency of nodes on subscripts.

(6) SUCESOR(#ENT), PLACE1(N), PLACE2(N) -- Compact representation of the graph. Gives successors of each node.

(7) PAR-CNT(N) -- For node i stores the count of its parents.

(8) #CYC, CYCLIST -- The list of recursive cycles found and deleted from the adjacency matrix.

(9) INCYC(N) -- For each node denotes the cycle to which it belongs. Zero entry means node does not belong to any cycle.

Outputs of SCHEDLER

(1) ORDER(IN) -- a vector of nodes sorted in their execution order.

(2) RANK(N) -- a vector which gives node i its rank in rank (i).

(3) #LOOPS -- number of loops in the test.

(4) DOTAB(#LOOPS)- the loop table which contains the following subfields:

(4.1) FIRST --- the sequence number of the node (i.e. its index in 'ORDER') which is the first node in the iteration scope.

(4.2) LAST --- the sequence number of the node which is the last node in the iteration.

(4.3) SUB# --- the subscript ID (i.e. index to FSUB).

Data structures used internal to SCHEDLR are:

(1) TPAR_CNT(N) --- temporary copy of PAR_CNT.

(2) SEQUENCED(N) --- for node i denotes whether it has been sequenced.

(3) TBNEST(TBNSIZ)-- list of subscripts to be nested in the present iteration.

(4) NEST(NESTSIZ) -- stack of subscripts in which the current subscript is nested.

(5) CHARAC(N) --- for node i denotes whether it is subscript free.

(6) IC --- name of the current subscript being trial ordered.

(7) IOC(#IOC) --- list of subscripts which got completely scheduled in the current trial.

(8) IOP(#IOP) --- list of subscripts which got partially scheduled in the current trial.

(9) #CYPART --- the number of recursive cycles which will be broken if the current trial subscript is scheduled.

(10) CP --- true means IC got completely scheduled, false means IC got partially scheduled.

SCHEDLR has 3 phases. In phase 1 it picks up all the source nodes and finds their subscripts* to form a candidate list (CANDLIST). If a source mode is subscript-free entry 0 is made in CANDLIST. It, then, forms best candidate list (BESTCANDLIST) out of the candidate list as follows.

Case (i) It first tries to find intersection of CANDLIST and TBNEST lists.

Case (ii) Otherwise, it tries to find intersection between CANDLIST and NEST stack.

Case (iii) Otherwise, it searches to see whether there is subscript-free in CANDLIST.

Case (iv) Otherwise, it includes all entries of CANDLIST.

It is described more fully in the Algorithm 5.9.

In phase 2 it calls on ORDERER with each of the subscripts in BESTCANDLIST to perform a trial schedule. The ORDERER called with the subscript IC returns the following information.

CP -- True if IC got completely scheduled.

IOC -- List of other subscript completely scheduled.

IOP -- List of other subscripts partially scheduled.

*In the Nopal system a subscript represents a range, and set of nodes which depend on the subscript is a range-set.

#CYPART -- Is set to greater than one if a recursive cycle cannot be scheduled completely.

It then calls on PRIOANAL to evaluate the results of the trial ordering. The priority is determined according to the Table 5.7.

If the priority is 1 it goes on to phase 3 otherwise it tries trial ordering for all the candidates in BESTCANDLIST and stores the one which yields the highest priority.

After all the candidates in BESTCANDLIST have been evaluated, error message is issued if the hightest priority was found to be 5, and a warning if it was 4. If priority is not 5 it goes on to phase 3 to do the actual scheduling, otherwise it returns.

The details of phase 2 are described in the Algorithm 5.9.

In phase 3 it calls the ORDERER again but this time to do the actual scheduling of nodes. This time the ORDERER schedules the nodes which have the subscripts IC and all the ones in NEST stack. It then updates ORDER, RANK, SEQUENCED and PAR_CNT.

Finally, if some nodes still remain to be scheduled it goes back to phase 1.

5.2.7.2 Procedure ORDERER

This procedure tries to order the nodes belonging to the subscripts specified by its inputs. It performs trial order if a flag TIMES is not set, and the actual ordering otherwise.

C/P ↓ I_c	I_{op}: CYPART	I_{op}=NULL =0	I_{op}= no I/O =0	I_{op} = I/O =0	I_{op} = I/O ≠0
C	I/O	1	2	4	5
	no I/O	1	2	3	5
P	I/O	4	4	4	5
	no I/O	2	3	3	5

Table 5.7 **Scheduling Priorities Based On Results Of First Trial Schedule**

ALGORITHM 5.9 SEQUENCING SCHEDLR

FUNCTION: Sequencing and iteration analysis.

CALLS: ORDERER, PRIOANAL (and Support Routines

ADD, DELETE, INTERSECT, STRTITER, ENDITER, BELONGS)

CALLED BY: INTSEQ

Step 1: Initialize SEQUENCED(N) to 0.

Initialize CHARAC(i) to 0 for node i if node i is subscript-free.

Step 2: Do the following 3 phases

Step 2.1 Phase 1

Step 2.1.1 For all nodes i which do not have predecessors (i.e. PAR_CNT(i) = o) include their subscripts in list CANDLIST. (Include zero in case there are nodes which do not have any subscripts)

Step 2.1.2 For all subscripts in CANDLIST get the "best" subscripts and put them in list BESTCANDLIST as follows:

Step 2.1.3 Case 1: For all subscripts in CANDLIST which are also in TBNEST put them in BESTCANDLIST Go to step 2.2

Step 2.1.4 Case 2: If there is an entry in the NEST stack which belongs to CANDLIST then repeat step 52.1.5 until the match found.

ALGORITHM 5.9 (continued)

Step 2.1.5. If top of NEST stack is a member of CANDLIST place it in BESTCANDLIST and go to 2.2. else pop NEST stack and generate end of iteration for the corresponding subscript.

Step 2.1.6. Case 3: If subscript-free range (i.e.0) is a member of CANDLIST put it in BESTCANDLIST and go to Step 2.2.

Step 2.1.7. Case 4: Put all the entries in CANDLIST into BESTCANDLIST.

Step 2.2 phase 2: For each of the subscripts in BESTCANDLIST do the steps 2.2.1 to 2.2.6. Let i be entry in BESTCANDLIST.

Step 2.2.1. If it was case 1 above push BESTCANDLIST(i) on NEST stack.

Step 2.2.2. Call ORDERER with argument false to do a trial schedule /* it produces IO, IOC IOP, #CYPART, CP, etc */

Step 2.2.3. Call PRIOANAL to analyze the results of ORDERER. Let the priority returned be P.

Step 2.2.4. If P = 1 then go to Step 2.2.9.

Step 2.2.5. Keep the value of the best priority so far in VALBESTSUB and the subscript name in BESTSUB.

Step 2.2.6. end of loop from 2.2.1.

Step 2.2.7. If VALBESTSUB = 5 then issue error message that a recursive - cycle has to be broken. Return.

ALGORITHM 5.9 (continued)

Step 2.2.8. If VALBESTSUB = 4 then issue a warning message that a virtual subscript must be made physical.

Step 2.2.9. Put all the overlapping subscripts as determined by ORDERER in the list IO into TBNEST list.

Step 2.3 Phase 3: /* actual scheduling */

Step 2.3.1: Call the ORDERER again this time with argument true to do the actual scheduling. /* ORDERER will now schedule by only the nodes with the subscripts IC and the ones on NEST stack. It will also update the global data structures, ORDER, RANK, SEQUENCED, PAR_CNT */

Step 2.3.2. If it was Case 2 and no new nodes got scheduled in this call to ORDERER, end the iteration corresponding to the subscript. Pop the NEST stack.

Step 3 If some nodes remain to be scheduled go to Step 2.

Step 4 Do step 4.1 until the NEST stack is empty.

Step 4.1. Pop the NEST stack and generate end of iteration for the subscript just popped.

Inputs to ORDERER are:

(1) TIMES -- Flag which tells whether actual ordering should be performed.

(2) NEST -- If TIMES is not set then it will pick the subscript on the top of the NEST stack as IC.

If TIMES is set then it picks up all the subscripts on NEST, and IC as before.

(3) It uses the global data structures ORDER, RANK, SEQUENCED, PAR_CNT, CHARAC, CYCLIST etc. described in the program SCHEDLR.

Outputs (If TIMES = false) of the trial order:

(1) CP -- true if all nodes belonging to IC got sorted.

(2) IOC(#IOC) -- List of subscripts which got completely scheduled.

(3) IOP(#IOP) -- List of subscripts which got partially scheduled.

(4) IO(#IO) -- Union of IOC and IOP.

(5) #CYPART -- Not equal to zero means that a recursive cycle was only partially scheduled.

Outputs (If times = true) of the actual order:

(1) ORDER(N) -- Add the new ordered nodes in it.

(2) SEQUENCED(N)-- Mark the new ordered nodes as sequenced.

(3) PAR_CNT(N) -- Decrement the parent count of the successors of the nodes which got ordered.

(4) RANK(N) --Enter the rank for the new nodes which got ordered.

(5) TPAR_CNT(N) -- Copy of PAR_CNT(N). Used in trial ordering.

Internal data structures:

(1) TORDER(#TORDER) -- Trial order vector

The ORDERER first gets the sbuscript on top of nest stack into IC. Nodes depending on it are to be ordered.

If TIMES is false it gets all the nodes corresponding to IC. It calls TOPOLOS which topologically sorts them, places the result in TORDER and RANKLIST and creates the temporary TPAR_CNT.

It thencalls TOPOANAL, does an analysis and finds the overlapping range-sets which got completely and partially scheduled;

and puts them in IOC(#IOC) and IOP(#IOP). It also finds whether IC got completely scheduled and sets CP as true. Finally, it checks to see whether the recursive cycles given by CYCLIST got completely scheduled. If so #CYPART is set equal to Ø. It then returns (to SCHEDLAR which evaluates the priority of the trial ordering using IC).

If TIMES is true it gets all nodes belonging to the subscripts of IC and all the elements of NEST stack. It again calls TOPOLOS which topologically sorts them, places the result in TORDER and RANKLIST and creates the temporary TPAR_CNT.

It then appends ORDER with TORDER, marks the nodes belonging to TORDER as done by setting corresponding entry of SEQUENCED as true. It also updates RANK and PAR_CNT as given by RANKLIST and TPARENT.

ALGORITHM 5.10: ORDERER

FUNCTION: Do an ordering for the nodes for the given subscripts.

CALLS: TOPOLOS, TOPOANAL, UPDATE

CALLED: SCHEDLR

Step 1: IC = top of NEST stack.

Step 2: If ¬TIMES then

Step2.1 Get all nodes which depend on the subscript of IC, and put them in NODESET (#NODES).

Step 2.2 Call TOPOLOS with NODESET as input arguments, and it returns the result in TORDER and RANKLIST.

Step 2.3 Call TOPOANAL to analyze the TORDER. It sets IO, IOC, IOP, CP, #CYPART as described earlier.

Step 2.4: Return.

Step 3: If TIMES then

Step 3.1: Get all nodes which depend on the subscripts IC and all entries of NEST.
Place them in NODESET.

Step 3.2: Call TOPOLOS with NODESET as input arguments, and it returns the result in TORDER and RANKLIST.

Step 3.3: Append ORDER with TORDER, and RANK with RANKLIST. Mark the nodes corresponding to entries in TORDER as SEQUENCED.
Copy TPAR_CNT into PAR_CNT.

Step 3.4: Return.

5.2.7.3 TOPOLOS:

This program performs the ranking and sequencing of the nodes given to it irrespective of their subscripts.

Inputs:

INLIST(#IN) -- Entries to be ranked

PAR_CNT(N) -- Parent count of each node in the adjacency matrix.

SUCESOR(#ENT), PLACE1(N), PLACE2(N) -- described in SCHEDLR.

CURANK -- Current value of rank.

OUTPUT:

OUTLIST(#OUT) -Entries INLIST which were ordered are placed in OUTLIST.

OUTRANK(#OUT) -Rank of the entries placed in OUTLIST.

TPAR_CNT(N) -- New Parent count.

Internal data structures

INFLAG(N) -- ith flag is set to indicate that INLIST(i) is ranked.

The algorithm used is the well known topological sorting and is described in Algorithm 5.11.

ALGORITHM 5.11 TOPOLOS

FUNCTION: To sort the given nodes irrespective of their subscripts.

CALLS: None

CALLED BY: ORDERER

Step 1: Copy PAR_CNT into TPAR_CNT.

Reset INFLAG(#IN) to false.

#TIN = #IN; #OUT = 0

Step 2: RANKLEVEL = CURANK

#NEW = 0

Step 3: Do Step 3.1 and 3.2 for all nodes i in INLIST.

Step 3.1 If INFLAG(i) is false and TPAR_CNT(INLIST(i)) = 0 then add it to OUTLIST and OUTRANK.

Step 3.2: #NEW = #NEW + 1

Step 4 If #NEW = 0 then return

Step 5 For all the nodes in OUTLIST(#OUT) to OUTLIST (#OUT + #NEW) decrement the TPAR_CNT.

Step 6 RANKLEVEL = RANKLEVEL +1;

#OUT = #OUT + 1;

#NEW = 0

Step 7 Go to Step 3.

5.2.7.4 TOPOANAL

This program analyzes the result of the trial ordering performed by ORDERER.

Inputs:

IC -- Name of the subscript being trial ordered.

TORDER(#TORDER) -- The trial order vector.

#NODES -- Number of nodes which were tried to be ordered i.e. which had the subscript IC.

OUTPUTS:

CP -- True if all the nodes with subscript IC got ordered. false otherwise.

IOC(#IOC) -- Subscripts all of whose nodes got scheduled.

IOP(#IOP) -- Subscripts some of whose nodes got scheduled.

#CYPART -- Non-zero means a recursive cycle had to be broken.

The algorithm used is described in Algorithm 5.12.

ALGORITHM 5.12 TOPOANAL

FUNCTION: To analyze the result of trial ordering done by ORDERER.

CALLS: None

CALLED: ORDERER

Step 1: Initialize #CYPART, #IO, #IOP, #IOC to zero.

Step 2: If #TORDER = 0 then #CYPART = 1;RETURN.

Step 3: If IC = 0 then Return.

Step 4: For each of the nodes in TORDER get its subscripts and place in the list IO.

Step 5: For each element J of the set IO do the following:

Step 5.1: Find all nodes depending on the subscript J. Let them be denoted by A.

Step 5.3: If A is contained in TORDER then enter J in the list IOC else enter J in the list IOP.

Step 6: For each node N in TORDER INCYC(TORDER(N)) $\neq \emptyset$ do the following: /* i.e. N occurs in a recursive cycle */

Step 6.1: If all members of the recursive cycle containing N are not in TORDER set #CYPART = 1. Return.

Step 7: Return.

5.2.8 Flowchart Report

This section briefly describes the report generated in the intra-test module analysis and sequencing phase of the NOPAL processor. The flowchart report consists of three parts: (1) an adjacency matrix, (2) a sequencing report, and (3) an iteration table. The adjacency matrix has been discussed in the previous sections (e.g. see Section 5.2.3). We will describe the interpretations of the latter two tables. A test module names GET_DATA is used throughout this section to illustrate the flowchart report. NOPAL source program of GET_DATA is shown in Figure 5.9.

This test applies stimulus to PIN1 and PIN2 of NAND gate, issues some informative messages to operator, and takes the measurement DIGITAL_RESP from PIN3. The measured value OUT is saved for subsequent use. There are 24 gates to be tested, and we want to apply three different patterns to each gate. The usage of the subscripted variables, e.g. IN(PAT), OUT(GATE, PAT), etc. implies a double loop involved in the test sequencing. The corresponding flowchart report for test GET_DATA is shown in Figure 5.10. We will frequently refer to statement n where n is an index to the order vector (first column of the sequencing report in Figure 5.10).

Statements 1 to 3 of the sequence report just declares SUBS is a global variable or function and GATE and PAT are two free subscripts. These are merely declarations and will not generate object code. LOOP-1 starts from statement 4 and ends at (inclusive) statement 5. This loop specifies that subscript PATTERN should iterate from 1 to 3 to fill input array IN(PAT).

```
16          TEST GET_DATA;
17              STIM;
18                  CONJ: <P1_12,P1_2> = P_SUPPLY(5.0 VOLT)  &
18                        <PIN1(GATE),PIN2(GATE)> = DIGITAL_STIM(IN(PAT))
18                                                 SOURCE: IN(PAT);
19                  ASSERT: GATE = SUBS('PIN1,PIN2,PIN3,OUT',24)
19                                                          TARGET: GATE;
20                  ASSERT: PAT = SUBS('IN,OUT:2',3)        TARGET: PAT;
21              MEAS;
22                  CONJ: <PIN3(GATE)> = DIGITAL_RESP(OUT(GATE,PAT))
22                                                 TARGET: OUT(GATE,PAT);
23              LOGIC: * FOR_GATE, * FOR_PAT;
24          DIAGNOSIS FOR_GATE:
24              PARAM = GATE,
24              TYPE = GATE_MSG;
25          MESSAGE GATE_MSG:
25              TEXT = 'TESTING GATE (P) ##;';
26          DIAGNOSIS FOR_PAT:
26              PARAM = (PAT,IN(PAT),GATE),
26              TYPE = PAT_MSG;
27          MESSAGE PAT_MSG:
27              TEXT = '      APPLYING PAT (P1)#, I.E. (P2) B''##'' ';
```

FIGURE 5.9: A TEST MODULE WITH SUBSCRIPT USAGE

```
                    INTRA MODULE SEQUENCING GET_DATA
                    ANALYSIS OF THE ADJACENCY MATRIX
                                                    1 1
                                1 2 3 4 5 6 7 8 9 0 1
                                - - - - - - - - - - -
  1  $S_W0001       CONJUNCTION 0 0 0 0 0 0 0 0 0 0 0
  2  $S_W0002       ASSERTION   0 0 0 0 0 0 0 1 0 0 0
  3  $S_W0003       ASSERTION   0 0 0 0 0 0 1 0 0 0 0
  4  $M_W0001       CONJUNCTION 0 0 0 0 0 0 0 0 0 0 1
  5  FOR_GATE       DIAGNOSES   0 0 0 0 0 0 0 0 0 0 0
  6  FOR_PAT        DIAGNOSES   0 0 0 0 0 0 0 0 0 0 0
  7  PAT            VARIABLE    1 0 0 1 0 1 0 0 0 0 0
  8  GATE           VARIABLE    1 0 0 1 1 1 0 0 0 0 0
  9  IN(PAT)        VARIABLE    1 0 0 0 0 1 0 0 0 0 0
 10  SUBS           VARIABLE    0 1 1 0 0 0 0 0 0 0 0
 11  OUT(GATE ,PA   VARIABLE    0 0 0 0 0 0 0 0 0 0 0

                    SEQUENCE OF PROCESSING FOR TEST GET_DATA

ORDER
VECT  ORDER   RANK      NAME          TYPE                    TEXT
INDEX VECTOR
  1    10      1       SUBS          VARIABLE        GLOBAL / SOURCE /

  2     2      2       $S_W0002      ASSERTION       GATE =  SUBS('PIN1,PIN2,PIN3,OUT',24)
                                                     TARGET:  GATE
                                                     SOURCE:  SUBS;

  3     3      3       $S_W0003      ASSERTION       PAT =  SUBS('IN,OUT:2',3)
                                                     TARGET:  PAT
                                                     SOURCE:  SUBS;

  LOOP-1 STARTS: SUBSCRIPT PAT ITERATES FROM 1 TO     3
  4     7      4       PAT           VARIABLE        LOCAL

  5     9      5       IN(PAT)       SUBSCRIPT_VA    GLOBAL / SOURCE /

  LOOP-1 ENDS;

  LOOP-2 STARTS: SUBSCRIPT GATE ITERATES FROM 1 TO    24
  6     8      6       GATE          VARIABLE        LOCAL

  7     5      7       FOR_GATE      DIAGNOSES       OTHER PARAMETERS=(GATE),  TYPE = GATE_MSG;

  LOOP-3 STARTS: SUBSCRIPT PAT ITERATES FROM 1 TO     3
  8     1      8       $S_W0001      CONJUNCTION     (<P1_12, P1_2>  = P_SUPPLY(5.0 VOLT )) &(<PIN1, PIN2>  =
                                                     DIGITAL_STIM(IN(PAT)))
                                                     SOURCE:  PAT,  GATE,  IN(PAT);

  9     6      9       FOR_PAT       DIAGNOSES       OTHER PARAMETERS=(PAT, GATE, IN(PAT)),  TYPE = PAT_MSG;

 10     4     10       $M_W0001      CONJUNCTION     (<PIN3>  = DIGITAL_RESP(OUT(GATE,PAT)))
                                                     TARGET:  OUT(GATE ,PA
                                                     SOURCE:  PAT,  GATE;

 11    11     11       OUT(GATE ,PA  SUBSCRIPT_VA    GLOBAL / TARGET /

  LOOP-3 ENDS;
  LOOP-2 ENDS;


LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS    4      LAST NODE IS    5      SUBSCRIPT IS PAT
   LOOP-2     FIRST NODE IS    6      LAST NODE IS   11      SUBSCRIPT IS GATE
   LOOP-3     FIRST NODE IS    8      LAST NODE IS   11      SUBSCRIPT IS PAT
```

FIGURE 5.10: FLOWCHART REPORT FOR GET_DATA

In fact, IN(PAT) is a global array and its elements are filled in some other test module. Consequently, LOOP-1 is just a dummy iteration. The code generator will not generate code for these nodes. LOOP-2 starts from Statement 6 and includes in its scope the rest of the statements in this test. In LOOP-2 and NAND gate is tested at a time. A message (Statement 7) is first sent to operator to indicate the gate number currently under examination. Then LOOP-3 starts to iterate three different patterns. In the double-subscripted range, i.e., Statements 8 to 10, each gate is applied a different input pattern (Statement 8) an information message is output (Statement 9); a measurement is taken for each gate under each pattern (Statement 10); and finally, output variable array OUT(GATE,PAT) is saved for later use. This concludes the description of sequencing table. An iteration table which summarizes all the loops involved in the test is given at the end of the flowchart report. Note that the node number is indexed to order vector, instead of adjacency matrix. Should there be no loop in the test, the summary table would not exist.

5.3 Inter-Test-Module Analysis and Sequencing

5.3.1 Overview of Inter-Test-Module Analysis

This section deals with the creation and analysis of the digraph for the whole module, and the determination of execution sequence of test modules. However, in the current implementation, since more than one test is allowed only in the MAIN module, this stage is used only for the MAIN module. The logical errors that may be detected during this phase are also identified. In inter-test analysis and sequencing, the relationships between three types of global objects are studied. These global objects are tests, diagnoses and global variables. In the analysis, the functions are treated as global variables which need not be targeted. Each of these objects may be subscripted. Just as a subscripted variable denotes multiple instances of the variable, a subscripted test or diagnosis denotes multiple instances of the test or diagnosis. A subscripted test or diagnosis is referred to as an "array" test or diagnosis. Just as with subscripted variables, the presence of the subscript in declaring a test or diagnosis causes the NOPAL processor to include the test or diagnosis in an iteration for the range of the subscript. Only a simple subscript

name may be used to qualify a test or diagnosis. Use of the subscript, e.g., TEST TEST1(I), indicates to the processor that the call to the test (TEST1, for example) is to be inserted within a loop on I.

There are seven kinds of relationships between the global objects in a specification. These precedence relationships with their recognition rules have been summarized in Table 5.1. Each precedence relationship corresponds to a row in the table. It is identified by a precedence type, and associated with a priority and 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. If there are no errors, the test modules are arranged in execution sequence. Loops are generated for array tests and diagnoses, so that these functions may be performed repetitively if so required.

5.3.2 Precedence Relationships

The seven sequencing strategies used in determining the precedence relationships are listed in Table 5.1 and explained below.

1) Data determinacy principle dictates that data must be generated before it can be used. The generation of data for a global variable by a predecessor test module is recognized

by its declaration as a TARGET variable. The use of a global variable by a successor test module is recognized when it is declared as SOURCE. A global variable has a diagnosis as a successor if it is used as a parameter in the diagnosis. The global variable has a diagnosis as a predecessor if the variable is used as an operator response in the diagnosis. This relationship is designated as type 1, and is mandatory. Therefore, it is associated with the highest priority 1. The priorities are 1 through 4, 1 being the highest and mandatory, and 4 being the lowest. The priorities 2 through 4 are not mandatory. This means the relationship may be ignored if it is so desired. This is the basis of cycle elimination for specifications.

2) Interactiveness relationships are dictated by the need to exchange messages interactively with the ATE operator. Its predecessor is a diangosis, 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 principle 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 of 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. It is designated as type 4, priority 1.

4) Fault isolation strategy schedules tests in a top-down fashion using component subset relationships. The more generic fault isolation tests are scheduled first. The lower level, more specific tests are then executed or skipped, depending upon whether or not 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) with priority 2 in this class. In type 5, the diagnosis D specifies a set of affected components in disjunction. For instance, if D is selected, then any (one or more) components in the affected component list (A or B or C) may have failed. In this case, a test module that may result in diagnosis of a subset of these components, say, A and/or B, is executed next to isolate the faults more specifically. On the other hand, if diagnosis D is not selected, then such a test module will not be scheduled for execution. In 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 all 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. If 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 strategy attempt to make efficient use of stimuli if they are already applied to the UUT. It is based on the assumption that application of stimuli are time-consuming, hence it is advisable to perform all the possible measurements, once a stimulus is available. Test modules which have frequently used stimuli triplets predecessors to modules which have less frequent stimuli triplets. This relationship is designated as type 9, priority 3.

6) Failure likelihood is based on the principle that timing efficiency is increased when those components which are more likely to fail are tested as early as possible. The failure rate index field in the UUT Component Failures specification provides this information. Hence a test module is a predecessor to another module if one of its affected components is more likely to fail than any one of the latters. The relationship is designated as type 10, priority 4.

7) In addition to the types of relationships discussed in the above six strategies, five other types, 11 through 15, exist between a test module and a diagnosis which is selected in the test module by the logical operators * (don't-care), |(or), |~(or-not), &(and), and &~(and-not) respectively.

The principle is that a diagnosis should be issued 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.

These relationships are obtained from the NOPAL specification and put into a weighted adjacency matrix as discussed in the subsequent subsections.

5.3.3 Creation Of The Weighted Adjacency Matrix

The rows (and columns) of the weighted adjacency matrix, W (corresponding to the digraph of the complete 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 5.11.

The steps of creating the weighted adjacency matrix are summarized in Algorithm 5.13. This is the same as Algorithm 5.1 except that the matrix represents different set of entitites. It first obtains the size, N, of the 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 entities, allocates the matrix W, and finally initializes W to all zeros (steps S2 to S4).

Now all types of precedence relationships as listed in Table 5.1 which exist in the NOPAL test specification can be entered into the weighted adjacency matrix, W. In the subsections 5.3.3.1 through 5.3.3.6 this process is described.

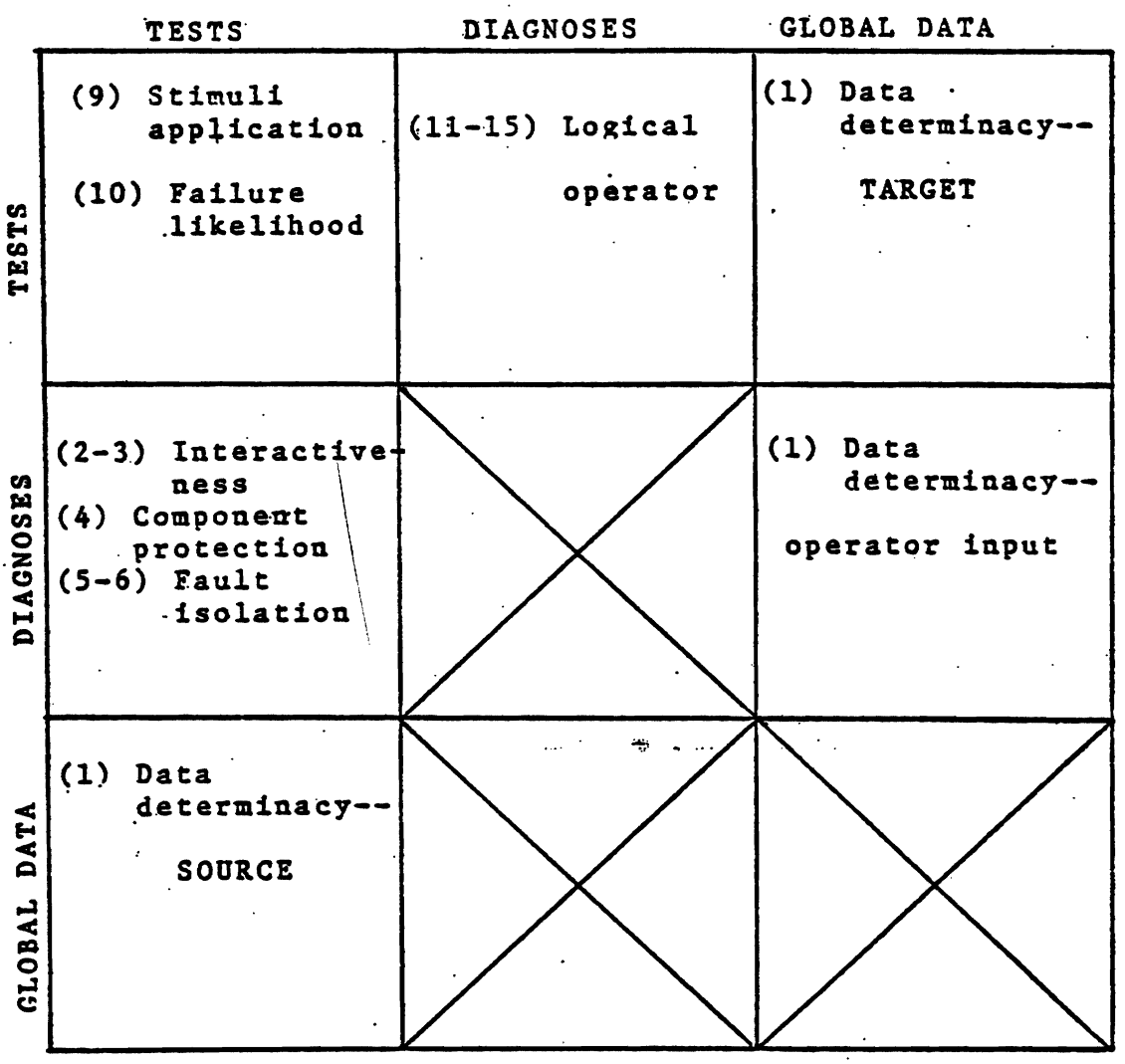


FIGURE 5.11: LAYOUT OF WEIGHTED ADJACENCY MATRIX

ALGORITHM 5.13: 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.

5.3.3.1 Data Determinancy Relationship

Data determinancy 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 5.14 shows how data determinancy relationships are detected and entered in the matrix W. Each test module t (corresponding to node i) in the NOPAL specification is examined for the presence of global variables (steps S1 to S5). For each TARGET variable v (corresponding to node j) in a waveform (i.e., conjunction or assertion), type 1 is entered into the matrix W in row i and column j to indicate that the test module is a predecessor and the variable a successor in the relationship of data determinancy (step S4). 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), the operator response variables (corresponding to node j) are examined to determine whether or not they are global. For each such global variable, type 1 is entered into W in row i and column j to denote that the diagnosis is the predecessor of the global variable.

ALGORITHM 5.14: DETECT AND ENTER DATA DETERMINANCY RELATIONSHIP

S0: Designate data determinancy relationship as type 1, priority 1.

S1: For each test module t in the NOPAL specification, perform steps S2 through S9.2.

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 global TARGET variable v in the waveform w, perform steps S4.1 to S4.4.

S4.1: Let j be the node number for the variable v.

S4.2: W(i,j) = 1.

S4.3: Enter the variable v in the GL_VAR array if not already in.

S4.4: If v occurs in a structure declaration enter it in VINH array. (To speed up processing in INTSEQ.)

S5: /* SOURCE global variables */
For each global SOURCE variable v in the waveform w, perform steps S5.1 to S5.4.

S5.1: Let j be the node number for the variable v.

S5.2: W(j,i) = 1.

ALGORITHM 5.14 (continued)

S5.3: Enter the variable v in the GL_VAR array if not already in.

S5.4: If v occurs in a structure declaration enter it in VINH array.

S6: For each diagnosis d in test module t, perform steps S7 to S9.2.

S7: Let i be the node number for d.

S8: For each global (operator response) variable v in d perform steps S8.1 through S8.2.

S8.1: Let j be the node number for v,

S8.2: W(i,j) = 1.

S9: For each global (other parameters) variable v in d perform Steps S9.1 through S9.2.

S9.1: Let j be the node number for v,

S9.2: W(i,j) = 1.

S10: Return.

5.3.3.2 Interactiveness and Logical Operator Relationship

Interactiveness relationships are entered between a diagnosis and a test module which are 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 5.15 shows the procedure to detect and enter these relationships. For each test module (corresponding to node i), the logic-diagnosis list is examined (steps S1 to S5). Depending on the logical operator in each logic-diagnosis pair, the corresponding 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 the matrix W in row j and column i to indicate that the diagnosis is a predecessor of the test module by the 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 a successor of the test module.

5.3.3.3 Component Protection Relationships

Component protection relationships (type 4) may exist between diagnoses and test modules. Algorithm 5.16 shows how this relationship is entered into the matrix W. First, for each affected component, a set T of the test modules which

ALGORITHM 5.15: DETECT AND ENTER INTERACTIVENESS AND LOGICAL OPERATOR RELATIONSHIP

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. All relationships have priority 1.

S1: For each test module t 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 to S5.

S4: Let j be the node number for diagnosis d.

S5: If op = '?' then set W(j,i) = 2;
else if op = '? ' then set W(j,i) = 3;
else if W(i,j) ≠ 0 then issue 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.

ALGORITHM 5.16: 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 Step 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 to 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 for each c */

S12: Return.

includes it as an affected component is obtained (steps S1 to S3). For each affected component in the component protection list of this affected component, another set D of diagnoses, each of which includes the original component as an affected component, is computed (steps S4 to S5). Finally, for each diagnosis (corresponding to node i) in the set 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 the diagnosis precedes the test module due to component protection relationship.

5.3.3.4 Fault Isolation Relationship

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. Otherwise, if it diagnoses components in conjunction, a type 6 is entered.

Algorithm 5.17 gives the steps for detecting and entering these relationships. For each test module t, a set C_t of distinct affected components in all the diagnoses of the test module t is computed (steps S1 to S2). Then, for each diagnosis (corresponding node i) another set D of affected components in this diagnosis is obtained (steps S3 to S5). 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 (ptype) is entered into

ALGORITHM 5.17: DETECT AND ENTER FAULT ISOLATION RELATIONSHIP

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 all 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 test-module 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.

the matrix W in row i and column j to indicate that the diagnosis is a predecessor of the test module.

5.3.3.5 Stimuli Application Relationship

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 test modules.

Algorithm 5.18 shows how these relationships are detected and entered to the matrix W. It begins by the counting of the occurrence stimuli triplets (steps S1 to S3.2). The count of the occurrence (frequency) of a stimuli triplet is the total number of occurrences of this triplet in all of the test modules. Next, for each test module t, the count of the stimuli triplet in t which has the highest number of occurrence is obtained. If t has no stimuli triplet, the count is zero (steps S4 to S5.2). Finally, if for any two test modules (corresponding to nodes i and j), the frequency of occurrence of the triplet in i is greater than the frequency in j, then type 9 is entered into the matrix w in row i and column j. If the frequency is less, then type 9 is entered in row j and column i (steps S6 to S11).

5.3.3.6 Failure Likelihood Relationship

Algorithm 5.19 shows the steps of detecting and entering the failure likelihood relationships (type 10) into the matrix W. This information is obtained from the failure-index field of each affected component in the specification of UUT component failures.

ALGORITHM 5.18: DETECT AND ENTER STIMULUS APPLICATION RELATIONSHIP

S0: Designate stimuli application relationship as type 9, priority 3.

S1: Initialize the list for stimuli triplets to null.

S2: For each test module t, perform step S3 to S3.2.

S3: For each stimuli triplet s in t perform steps S3.1 to S3.2.

S3.1: If s does not appear in the list of distinct stimuli triplets, then add it to the list and set its frequency count field to 0,

S3.2: Increment frequency count field of s by 1.

S4: For each test module, perform steps S5 to S5.2.

S5: Let i be the node number for t.

S5.1: If t contains no stimuli triplets then set INDEX(i) = 0,

S5.2: If t contains a stimuli triplet then set INDEX(i) = count of the most frequent stimuli triplet of t.

S6: Let #TESTS be the total number of test modules.

S7: For i=1 to (#TEST-1), perform steps S8 to S11.

S8: For j = (i+1) to #TESTS, perform steps S9 to S11.

S9: If INDEX(i) < INDEX(j), then set w(j,i) = 9 else set w(i,j)=9.

S10: End loops j and i.

S11: Return.

ALGORITHM 5.19: 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 failure-index in the test module */

For each test module t, 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 the lowest failure index in 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 S14.

S9: If FAIL_INDICES(i) = 0, then go to Step S14.

S10: For j = (i+1) to #TESTS, perform steps S11 to S14

S11: If FAIL_INDICES(j) = 0, then go to step S14.

S12: If FAIL_INDICES(i) < FAIL_INDICES(j) then if W(i,j) = 0, then set W(i,j)=10.

S13: If FAIL_INDICES(j) < FAIL_INDICES(i) then if W(j,i)=0, then set W(j,i)=10.

S14: End loops j and i.

S15: Return

For each test module the algorithm obtains the failure index of the affected component which has the lowest failure index in the test module (steps S1 to S6). Then for each pair of test modules (corresponding to nodes i and j), if the lowest failure index of the affected components in test module i is lower than that in test module j, then 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 obtained 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 will not proceed until the user resubmits the NOPAL test specification after correcting all the errors identified until this phase.

5.3.4 Graph Analysis of Adjacency Matrix

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 5.2 shows the weighted adjacency matrix for the sample test specification MINIRADIOSET of Figure 4.7.

The task of this phase is to perform further analysis on the weighted adjacency matrix to detect possible logical errors. These errors relate to the incomplete, ambiguous, or inconsistent use of the global objects.

This section describes the preliminary analysis of W where the more obvious errors are detected. The next section 5.3.5 is about the less obvious errors due to circular definition or use of global objects.

Two types of errors are detected during the preliminary analysis: (1) the errors that arise between global variables and tests or diagnoses, and (2) errors that arise between tests and diagnoses. In the first case, the following three conditions are detected: (1) the use of an undefined global variable, (2) the definition of a global variable more than once, and (3) the definition of a global variable which is never used. The first condition is an error, while the last two are only warnings.

(1) to detect an undefined global variable v (corresponding to node i), the entries in column i of W are checked. If all rows of column i are zeros, then the variable v has never been defined, so error message #1 is issued.

(2) to determine whether or not a variable v (corresponding to node i) has been multiply defined, the entries in column i are checked. If there is more than one non-zero entry, the variable v has been targeted more than once. Hence warning message #3 is issued.

(3) To detect if a global variable v (corresponding to node i) has been defined but never used, row i is searched for all zero entries. If this is the case, warning message #2 is issued.

To detect errors between tests and diagnoses, only two conditions are searched: (1) a test with no diagnosis, and (2) multiple selection of tests by a diagnosis through the logical operators after (A) and after-not (A~). The first condition is only a warning, but the second one is an error.

(1) To determine if a test (corresponding to node i) no diagnosis, all entries in row i and in columns corresponding to diagnoses are checked. If all the entries are zeros, then no diagnosis is connected to the test module i by the logical operators *, |, |~, &, &~. Then the entries in column i and in rows corresponding to diagnoses are checked. If there is no entry of type 2 or 3, then there is no diagnosis connected to the test module by the logical operators A and A~. If both of the above conditions are true, the test module has no diagnosis. Hence warning message #8 issued.

(2) To detect multiple selection of tests by a diagnosis (corresponding to node i) through the logical operators A and A~, the multiple occurrence of either type 2 or type 3 entry in row i is checked. If either type 2 or type 3 occurs more than once, error message #15 is issued:

This completes the preliminary analysis of the weighted adjacency matrix.

5.3.5 Sequence Determination

5.3.5.1 Subscript Processing

In order to facilitate sequence determination for subscripted global objects, an alternate representation of the array graph is used in the scheduling phase. Each global object (test, diagnosis, or variable) is considered to be a node. Each node has a subscript list attached to it. In addition, each edge in the array graph has a subscript expression list attached to it. There are two types of subscript expressions used in external sequencing. A Type 1 subscript expression is a simple subscript reference, such as in & DIAG1(I). Type 1 subscript expressions appear on edges drawn from tests to diagnoses and between global variables and other global objects. Type 2 subscript expressions appear on edges inserted by the system to ensure that tests and their associated diagnoses will be placed in the same loop. For each edge from an array test to a diagnosis selected by the test's LOGIC statement, the system inserts an edge from the diagnosis to the test with a Type 2 subscript expression for each subscript.

The data structure of the node subscript list is as follows:

```
01 node_sublist based (p_nsublist),
   02 next_sub ptr,
   02 node_subid fixed bin, /*index in gl_sub*/
   02 idwith fixed bin; /*identified with level of loop*/
```

The data structure for the edge and the edge's subscript expression list are as follows:

```
01 edge based (edge_ptr)
   02 next_edge ptr,
   02 target fixed bin,
   02 edge_type fixed bin,
   02 dimdif fixed bin,
   02 subx ptr; /*subscript expression list*/

01 edge_subl based (edge_sublptr),
   02 nxt_subl ptr,
   02 local_sub$ fixed bin, /*ordinal pos of this sub in target*/
   02 apr_mode fixed bin;
```

Prior to determination of execution sequence, consistency checks on the use of subscripts is performed. If a test is subscripted, each diagnosis referenced in the

test's LOGIC statement must have the same subscripts. There is one exception to this rule. If a diagnosis is selected with after (?) or after-not (?^) logic, then that diagnosis need not be subscripted. However, if a selected diagnosis is subscripted, its subscript(s) must match the test's subscript(s). Each subscript expression used in referencing a test or diagnosis must be of Type 1.

If a subscripted global variable is used in a test, its subscript list must be a subset of the test's subscript list. That is, the variable must have at least the same subscripts as the test. It may have more. The subscript expressions for the variable's global subscripts must be of Type 1.

SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION MINIRADIOSET

ORDER VECTOR INDEX	ORDER VECTOR	RANK	TYPE	NAME
1	1	0	TEST	DC_INPUT
2	7	1	DIAGNOSIS	D2
3	8	1	DIAGNOSIS	D3
4	5	2	TEST	FREQ
5	16	3	DIAGNOSIS	D4
6	17	3	DIAGNOSIS	D5
7	18	3	DIAGNOSIS	D6
8	20	3	GLOBAL VARIABLE	V1
9	2	4	TEST	AMPL
10	3	5	TEST	DISTORT_2W
11	4	5	TEST	DISTORT_VOLT
12	9	5	DIAGNOSIS	D7
13	10	5	DIAGNOSIS	D8
14	12	6	DIAGNOSIS	30
15	13	6	DIAGNOSIS	24
16	14	6	DIAGNOSIS	25
17	15	6	DIAGNOSIS	26
18	6	7	TEST	DISTORT_10MW
19	11	8	DIAGNOSIS	27
20	19	8	DIAGNOSIS	28

Figure 5.13: INTER-TEST MODULE SEQUENCING REPORT

5.3.6 Global Subscripted Variables

A specification which uses subscripted global variables inconsistently, (i.e. in some statements a variable is considered an array, and in others a scalar or an array of a different dimension) requires special consideration. Further, even if a user uses the number of dimensions of a variable consistently, there is a secondary problem of extracting from the specification the bounds for each dimension of a variable that is an array.

When we are determining the number of unique global variables and their names for the inter-test sequencing phase of the Processor, we also determine whether a global variable is a scalar or an array, and in the case of an array, the number of dimensions. While we are finding whether a global variable is being used as a scalar or an array throughout a specification, we perform error/warning checks in the context of using the variables consistently, i.e. if a variable v is used as a scalar or array once in a specification then it should be used throughout the specification in the same way, and in the case of an array, the same number of dimensions should be used. This process is described in Algorithm 5.20.

In the case that a global variable v is an array, such that the number of dimensions of v is uniform throughout the specification, then we wish to have a method to determine the bound of each dimension. This is accomplished through analyzing the SUBSCRIPT or SUBS function in the specification. This function has the following simplified syntax:

ALGORITHM 5.20 GLOBAL VARIABLE ATTRIBUTES DETERMINATION

S0: Initialize global variable list, GL_VAR, to null, and #VARS, number of variables on list, to 0, where GL_VAR has 3 fields: ID: variable identifier name, ARRAY: if equal to 1 then array, else if equal to 0 then scalar, and DIM#: if ARRAY is set to 1 then number of dimensions of v.

S1: For every use of a variable v in a specification, perform steps S2 to S10.

S2: If scope bit of v is marked global, then perform steps S3 to S10, else skip to step S10.

S3: If v is not on global variable list, GL_VAR, then perform steps S4 to S4.2, else skip to step S5.

S4: #VARS = #VARS+1; ID(#VARS) = v;

If v is a scalar perform step S4.1, else skip to step S4.2.

S4.1: ARRAY(#VARS)=0; /* v is scalar */

Skip to step S10.

S4.2: ARRAY(#VARS)=1; /*v is array */

DIM#(#VARS)=number of dimensions in this use of v.

Skip to step S10.

S5: Let i be the position of v on GL_VAR.

Perform steps S6 to S10.

S6: If v is used as a scalar in this use of v, and ARRAY(i) = 0 then skip to step S10, else skip to step S9.

S7: If v is used as an array in this use of v, and ARRAY(i) =1 then perform step S7.1, else skip to step S9.

S7.1: If number of dimensions in this use of v is equal to DIM#(i) then skip to step S10, else perform step S8.

ALGORITHM 5.20 (continued)

S8: ERROR: Global variable v is used as an array but with differing number of dimensions. Initially used with DIM#(i) number of dimensions. (Message # 33)

Skip to step S10.

S9: ERROR: Global variable v is used as an array and scalar, it cannot be both. (Message # 34)

S10: End of loop on each use of a variable v.

S11: Return.

< subscript variable = SUBSCRIPT(< parent list >,< bound >) ...

< parents list >::='< identifier >[:< integer >],[,< identifier>
[:< integer>]]*'

<bound >::= < integer >

(where [ ]* means repeats 0 or more times)

In terms of the scope of inter-test sequencing, this function is treated as the only vehicle to express the bounds on each dimension for every global variable array. The way this is realized is delineated in Algorithm 5.21. The main thrust of the algorithm is that, if there exists a global variable that is an array, the bounds of each dimension of the array should be available in an SUBSCRIPT function having this array as a parent name.

After completing these two checking procedures, the dimension and bound on each dimension for every global variable is obtained. This process makes easier the code generation phase of the processor, by way of the fact that these checks for the consistency of use of global variables are performed in Inter-test, which permits the former phase to avoid the issue of consistency.

ALGORITHM 5.21 GLOBAL SUBSCRIPTED VARIABLE BOUNDS DETERMINATION

S1: For each test module t in the NOPAL specification, perform Steps S2 to S9.

S2: Foreach SUBSCRIPT function appears in a simple assertion in t then perform Step S3, else skip to step S10.

S3: Set, PARENTS = parent list in SUBSCRIPT function, BOUND = bound in SUBSCRIPT function.

S4: For each parent in PARENTS, perform Steps S5 to S8.

S5: If parent is a global variable then perform step S6, else skip to Step S8.

S6: If the corresponding dimension of parent is undefined in the global variable list (i.e. equal to 0), then set it to BOUND, else verify that BOUND is equal to prior definition. If not then issue error message #12.

S7: If dimension of parent does not exist as a dimension of a global variable in the list, then there is inconsistency in the use of the parent list. Issue warning message #1.

S8: End of loop on each parent in PARENTS.

S9: End of loop on each SUBSCRIPT function int.

S10: End of loop on each test module t.

S11: Verify that the upper bounds for each dimension of the global variables are nonzero. In case there are zero entries issue error message #2.

S12: Return.

5.3.7 External Scheduling

The scheduling is done by two mutually recursive procedures SCHEDULE_GRAPH and SCHEDULE_COMPONENT. Both receive as a parameter a subgraph of the graph representation of the specification and produce a schedule as a result. Following is a schematic description of their actions.

5.3.7.1 Schedule_Graph

Accepts as parameters a graph G and a nesting level l. It decomposes G, using the procedure STRONG into maximal strongly connected components $G_1, G_2, \ldots, G_n$ already sorted according to their dependency order. For each G_i it calls on SCHEDULE_COMPONENT to produce a schedule for G_i, i.e.

$$S_i = \text{SCHEDULE\_COMPONENT}(G_i, l) .$$

It then returns the final schedule

$$S = S_1, S_2, \ldots, S_n$$

i.e., the ordered list of the schedules $S_1, S_2, \ldots S_n$.

5.3.7.2 Schedule_Component(G,l)

Accepts as parameters a graph G which is known to be a maximal strongly connected component and a nesting level l. For each node n contained in G, we count how many local subscripts have not yet been identified with assigned loop variables. Let MINFREE be the minimum number of available local subscripts over all the nodes in G.

If MINFREE = 0 and $|G| = 1$ then the graph consists of a simple nonrepeating node. The resulting schedule consists then of this simple node.

If MINFREE = 0 and $|G| > 1$ then try to delete low-priority edges. If this can be done, call SCHEDULE_GRAPH with the new graph. Otherwise, there is an error and a report is issued. The graph cannot be scheduled.

Otherwise we try to find a candidate identification for a loop variable. Such an identification consists of a local subscript position P(n) for each node n contained in G such that

(1) The P(n) local subscript in node n, $I_{p(n)}$, is still available.

(2) In any edge $N(I_m,\ldots,I_1) \leftarrow (J_s,\ldots,J_1)$ where the J_i are expressions involving $I_1,\ldots,I_m$ we require that $J_{p(m)} = I_{p(n)}[-c]$ i.e., the position corresponding to the loop variable is consistent in all edges. Since in the assertion or dependency corresponding to this edge the loop variable is to be identified with with $I_{p(n)}$, we require that $I_{p(n)}$ occupies the position allocated to the loop variable in M.

If we cannot find a candidate identification satisfying (1) and (2), we try to delete low priority edges. If this can be done, we call SCHEDULE_GRAPH with the new graph. Otherwise, the graph cannot be scheduled and we issue an error message.

Otherwise we set all the IDWITH fields of the local subscripts $I_{p(n)}$ for each n contained in N to l thus noting that these subscripts have been identified with the loop variable at level l. We also remove all edges of the form

$$N(I_m,\ldots,I_1) \leftarrow (\ldots I_{p(n)} - c\ldots) \quad \text{for} \quad c > 0$$

since they imply dependency on values from the previous iteration of the same loop. Denoting the graph thus modified by G^l, which may have ceased to be strongly connected by the removal of the edges, we call S = SCHEDULE_GRAPH(G^l,&+1) to Schedule G^l. Let us denote the newly introduced loop variable by v_l. Then the schedule returned by the currect procedure is: for v_l do S end $\{v_l\}$.

When the program will be generated, all the subscripts whose IDWITH field has been set to l will be replaced by v_l.

5.3.7.3 Representations

A graph is represented by a list of elements of type GNODE, each having the following fields:

NXT_GNODE--A pointer to the next element in the list.

NODE_ID--The node number (in the directory) of the element.

SUXL--Pointer to a list of edges connecting this element to its successors. Initially this is identical to the SUCC-LIST list. But as the process proceeds some of these edges are removed from this list.

A strongly connected component is represented by the structure COMP having the fields:

NXT_COMP--Pointing to the next component.

NODE_LIST--Pointer to a graph which comprises the component.

A schedule is a list of schedule elements each of which is either a node-element or a for-element.

A node-element is declared as a structure NELMNT having the fields:

NXT_NLMN--Pointer to the next element in the schedule.

NLMN_TYPE--An integer, always equal to 1 for node elements.

NODE#--The node number.

A for-element is declared as a structure FELMNT having the fields:

NXT_FLMN--Pointer to the next element in the schedule.

FLMN_TYPE--Always equal to 2, denoting this is a for-element.

ELMNT_LIST--Pointer to a schedule which is the scope of the for-loop.

FOR_NAME--The node number of the loop variable which can be a FOR_EACH.X and then FOR_NAME is the nude numberof X, or it can be a declared subscript.

FOR_RANGE--The node number specifying the range of the loop variable in the loop.

5.3.7.4 The Main Program of Schedule

One of the basic processes in the procedure is that of finding maximal strongly connected components. We follow Tarjan's algorithm based on depth first search as described in Aho, Hopcroft and Ullman's book, "The Design and Analysis of Algorithms". The main part of the algorithm is the recursive procedure SEARCHC(V) which is presented with a node V and identifies all the strongly connected components reachable from V. It utilizes the arrays DFNUMBER, LOWLINK which are preset to Ø for all nodes, the global variable COUNT and a stack called STACK. In our implementation we also employ an array ONSTACK which is positive for node n if it is currently on the stack. STRONG(G) is a function which accepts a graph as a parameter and returns a sorted list of its strongly connected components. The procedure SCHEDULE_GRAPH has been described above.

5.3.7.5 Finding Strongly Connected Components

The main body of SCHEDULE starts by constructing a graph, i.e., a linked list of structures of the type GNODE representing the complete specification. Each I = 1,...,DICTIND is allocated an element sturcture with its NODE_ID field equal to I and its SUXL field pointing to a copy of the list SUCC_LIST(I). We also set the array NODEP(I) to point to the graph's element. This is necessary since the edges in SUXL refer to node numbers which we should translate to graph's elements. We then call once:

FLOWCRT = SCHEDULE_GRAPH(MAING,1)

where MAING is a pointer to the complete graph, 1 is the initial level, and FLOWCRT is a global pointer to the schedule which later procedures use to retrieve the schedule.

ALGORITHM 5.22: SEARCHC(V)

1. COUNT: = COUNT+1, DFNUMBER(v), LOWLINK(v): = COUNT.
   Put v on the stack.
2. Repeat the following substeps for each node w a direct descendant of v.
   2.1 If DFNUMBER(w)=o this is a new node not searched before. We call SEARCHC(w) and then let LOWLINK(v)=min (LOWLINK(v), LOWLINK(w)).
   2.2 Else, if DFNUMBER(w)>o and w is on the stack, then let LOWLINK(v)=min (DFNUMBER(w), LOWLINK(v)).
3. If LOWLINK(v)< DFNUMBER(v) return.
4. Else, LOWLINK(v)=DFNUMBER(v) and this is a root of a strongly connected component. All the elements (above and including v) on the stack are successively unstacked and linked together into a list - a subgraph which is defined as a component. This component is placed at the head of a list of components pointed to by the variable COMP_LIST. In addition we maintain a running component number COMP_CNT and set the array COMP#(w)=COMP_CNT for each w in the current component.

Note that the algorithm returns a list of components which are ordered consistently with the dependency order.

ALGORITHM 5.23: STRONG(G)

1. Clear the stack, the component count, the list of components and the variable count. For each $v \in G$ set

   DFNUMBER(v)=o

2. For each veG such that DFNUMBER(v)=o call SEARCHC(v) to add the components reachable from v to the top of the component list.
3. Delete from the graph all the edges which connect nodes in different components.
4. Return as a result the component list.

ALGORITHM 5.24; SCHEDULE_COMPONENT(G,1)

1. For each node n contained in G, compute the number of free local subscripts. These are local subscripts whose IDWITH = 0 which implies that they have not yet been identified with any loop variable. Let MINFREE be the minimal number of free subscripts over all n contained in G.
2. If MINFREE = 0 and |G| = 1 we return a schedule of one node element containing the single node in G. Exit.
3. If MINFREE = 0 and |G| > 1, then try to delete low-priority edges. If this is possible, call SCHEDULE_GRAPH with the new graph. Otherwise there is an error. The message: SCHEDULE: A CYCLE DETECTED is printed and then the procedure PRINT_CYCLE is called to print the remaining cycle. Return an empty schedule and exit.
4. Otherwise we have to search for candidate identification. We start by constructing in the array stack (denoted here by S) a list of the graph nodes such that for every i > 1 the node S[i] has an edge incoming from some S[j] , j < i . This is done by the following iterative process:
   4.1. Let S[1] be the first node in G. Let I = 1 .
   4.2. Repeat the following steps as long as I < |G|.
      4.2.1. Let n:=S[I].
      4.2.2. For each descendant of n which is not already on S, add it to S.
      4.2.3. I:=I+1, return to 4.2.1.
5. Let IDF be the node S[1]. Let POS range over all the free subscripts of IDF. Repeat steps 6-13 for each available subscript.
6. Clear POSITION for all nodes in the graph, and then set POSITION [IDF]:=POS.
7. Repeat steps 8-12 for I=1 to |G|.

8. Let n:=S[I], POST=POSITION(n) and consider each edge from node n to any other node t.

$$t(I_m,\ldots,I_1) \leftarrow n(E_s,\ldots,E_1)$$

Consider the subscript expression E_{POSJ} which corresponds to the identified position in n.

9. If E_{POSJ} is not a simple expression (I_j[-c]) or if the subscript is reduced then POSJ cannot be identified with a loop variable for the strongly connected component, exit to step 14 to consider the next value of POS.

10. If $E_{POSJ} = I_j$[-c] for some $\lambda < j \leq m$, check if POSITION [t] > 0 . If POSITION [t] > 0 and POSITION [t] $\neq$ j there is a conflicting identification in the subscripts of t. Exit to Step 14.

11. If POSITION [t] = 0 set POSITION [t]:=j.

12. Return to step 8 for the next I.

13. Arriving here means that a complete identification was successfully performed. Go to step 16.

14. The identification starting with position POS for node IDF has failed. If another free subscript for IDF is available set POS to it and return to step 6.

15. Arriving here means that no identification is possible. Try to delete low-priority edges. If this is possible, call SCHEDULE_GRAPH with the new graph. Otherwise, the message, "SCHEDULE: NO CANDIDATE SUBSCRIPT IN CYCLE" is printed, followed by a list of the nodes in the graph. Return the empty schedule and exit.

16. A successful identification! We proceed to determine name and range for the loop variable and to delete edges which are of the form

$$A(,\ldots I_p,\ldots,I_1) \leftarrow B(,\ldots I_p\text{-}c,\ldots)$$

where p is the identified position of A.

17. Allocate a for-element for the schedule with empty FOR_RANGE, FOR_NAME fields.

18. Scan each node in the graph, n contained in G. Let p=POSITION[n].

19. Examine the local subscript I_p . Set its IDWITH field to 1, the level parameter. If I_p has a range and FOR_RANGE is empty yet set FOR_RANGE to the range of I_p.

20. If FOR_RANGE has previous value which is different than the range if I_p, print the following warning message:

    SCHEDULE: A RANGE CONFLICT IN NODE node-name BETWEEN THE ALREADY ASSIGNED RANGE: $range_1$ AND THE NEWLY IMPLIED RANGE: $range_2$

    "node_name" is the name of node n.

    "$range_1$" and "$range_2$" are respectively node names whose range is assumed by the loop variable and the local subscript I_p.

21. If I_p has a range and FOR_NAME is empty yet assign to FOR_NAME the name associated with I_p.

22. Delete from the graph any edge of the form

$$t(\quad) \leftarrow n(E_s,\ldots,E_p,\ldots,E_1)$$

    where E_p is of the form $I_k - c$ for some k and $c > 0$. These correspond to dependencies on values created during the previous iteration and hence should be ignored.

23. Repeat steps 18 to 22 for all nodes in the graph.

24. Call SCHEDULE_GRAPH (G,ℓ+1) to further schedule the component. Set the field ELMNT_LIST of the for-element to the schedule returned by SCHEDULE_GRAPH. Return as a result the for-element.

    The following subprocedures are defined with SCHEDULE_COMPONENT:

    PRINT_CYCLE: Prints the names of all the nodes in the current component.

    CONCATENATE (A,B): Concatenate the list B to the end of the list A. A and B are pointers to general lists.

    FREE_PPAIR_LIST (LIST): Frees the space allocated to a list of PPAIR structures pointed to by LIST.

5.3.8 Flowchart Report

To give an overall view of the skeleton of the main routine to be generated, a flowchart report is prepared. The flowchart report has the following six sections: (1) declaration of global variables, (2) declaration of evaluation and control functions, (3) declaration of stimulus functions, (4) declaration of measurement functions, (5) declaration of failure functions, and (6) calling sequence of the test modules. The test module calls are listed according to their ranks which are identified through comments "/*EXECUTE LEVEL n*/". Sometimes, there will be several test module calls at the same level. This indicates that the execution sequence of tests can be chosen arbitrarily. Loops around array tests and their associated diagnoses are indicated. To simplify the flowchart report, the control logic preceeding each test module call is omitted. Such details can be found in the code generated.

The flowchart report for the radioset example is given in Figure 5.14. The first test to be performed is DC-INPUT. At the next level FREQ (rank 2), then AMPL (rank 4) are performed. At rank 5, there are two possible tests (DISTORT_2w and DISTORT_VOLT). Any one of the two can be executed first. Finally, DISTORT_10MV is performed. This completes the flowchart report from inter-sequence analysis.

```
                    FLOWCHART REPORT FOR NOPAL SPECIFICATION MINIRADIOSET

/***** FLOWCHART FOR GLOBAL PROCEDURE MINIRADIOSET *****/

  BEGIN MINIRADIOSET;

        DECLARE AS GLOBAL VARIABLES:
             V1;

        DECLARE AS EVAL OR CONTROL FUNCTIONS:
             /* NO EVAL OR CONTROL FUNCTIONS */

        DECLARE AS STIMULUS FUNCTIONS:
             SIGNAL_AM, SAM, CONST_S;

        DECLARE AS MEASUREMENT FUNCTIONS:
             SINE_D, SINE_DELAY, DISTORTION, CONST_R;

        DECLARE AS FAILURE FUNCTIONS:
             FREQ_TOL, REF_VOLT, DISTORT, AMPL_TOL;

      /*  EXECUTE LEVEL     0 */
        PERFORM TEST DC_INPUT;

      /*  EXECUTE LEVEL     2 */
        PERFORM TEST FREQ;

      /*  EXECUTE LEVEL     4 */
        PERFORM TEST AMPL;

      /*  EXECUTE LEVEL     5 */
        PERFORM TEST DISTORT_2W;
        PERFORM TEST DISTORT_VOLT;

      /*  EXECUTE LEVEL     7 */
        PERFORM TEST DISTORT_10MW;

  END MINIRADIOSET;
```

FIGURE 5:14: FLOWCHART REPORT OF THE INTER-TEST MODULE SEQUENCING FOR THE MINIRADIOSET SPECIFICATION

CHAPTER 6

CODE GENERATION

6.1 Overview

Code generation phase of the NOPAL processor accepts the weighted adjacency matrix, the order vector and several other data structures from the sequencing phase; and produces a complete object program in ATLAS. The code generation takes place in 3 stages illustrated in Figures ~~6.1~~ 3.7 and 3.8.

Stage 1: Declarations and diagnoses generation stage.

Stage 2: Intra-test code generation stage.

Stage 3: Inter-test code generation stage.

In stage 1 the procedure CDEMAIN (called from EXTSEQ) generates declarations for the system defined and the user defined global variables, declarations for the connection points, and the procedure definitions for each of the diagnoses in the NOPAL specification.

In stage 2 the procedure CDETEST (called from INTSEQ for each of the test modules) generates procedure definitions for the test modules. The procedure definition for each of the test modules contains code for waveforms and assertions in the test module (in the order determined by INTSEQ), and then logic for selecting the diagnosis (by means of calls on diagnosis procedures).

Finally, in stage 3, TRMNATE (called from SEQANAL) generates the code for performing tests i.e. code for calls on the test module procedures. TRMNATE also generates the termination code for the ATLAS program. ~~This stage is performed only for the main module. Other modules have only one test per modfun and hence this stage is not needed.~~

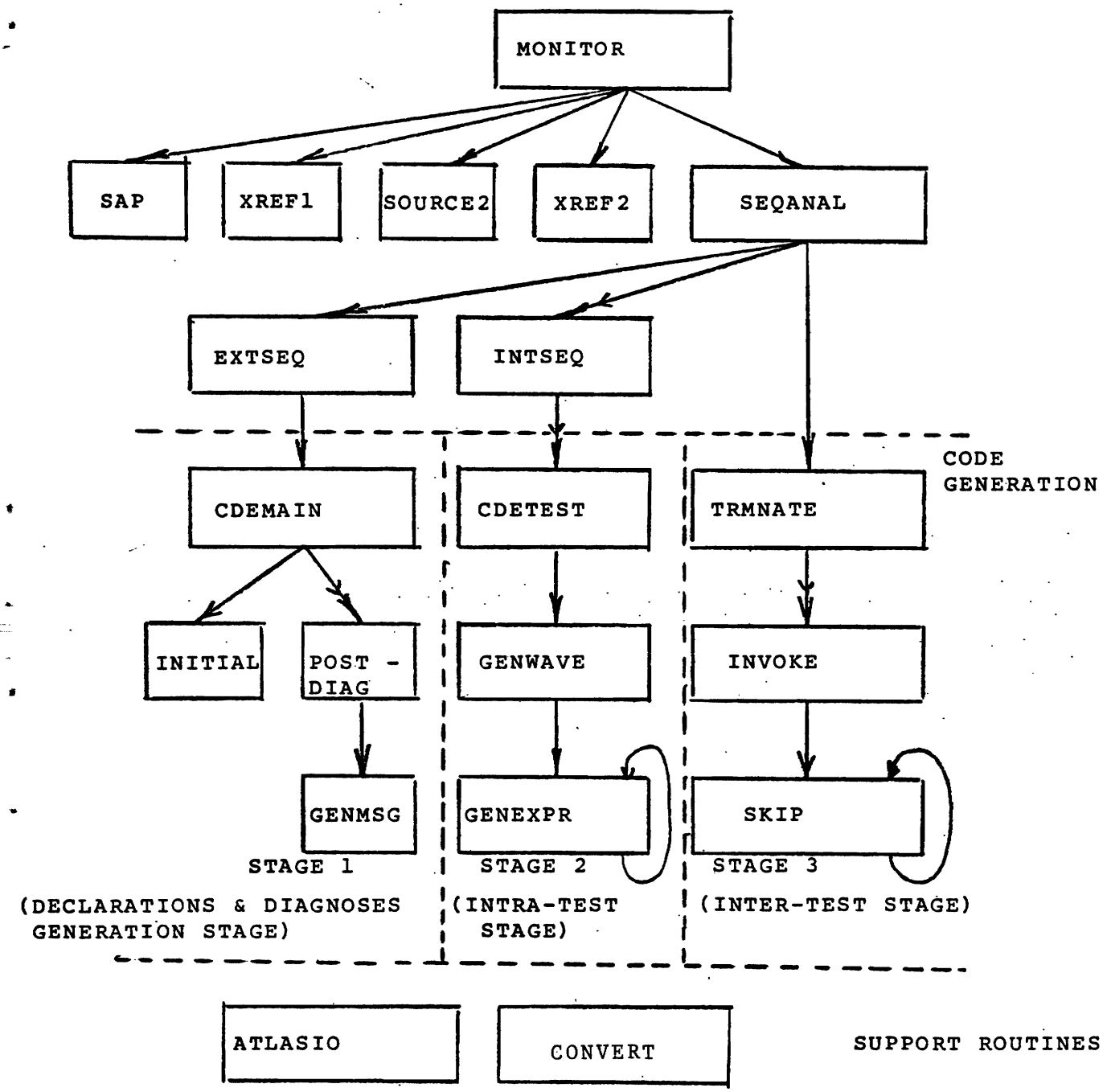


FIGURE 3.1 ~~6.1~~: PROGRAM CALLING STRUCTURE
(PRINCIPALLY SHOWING CODE GENERATION)

LEGEND:

1. A ⟶⟶ B: means that proc. A MAKES MULTIPLE CALLS ON B
2. A ⟶ B: means A makes a single call on B.

HEADER OF THE EQUATE-ATLAS PROGRAM

IDENTIFICATION
TEST MODULE NAME VS. ATLAS SUBSCRIPT

DECLARATION OF SYSTEM DEFINED VARIABLES

DEFINITIONS OF CONSTANTS

UUT POINT DEFINITIONS

DECLARATION OF USER DEFINED GLOBAL VARIABLES

USER DEFINED ATE FUNCTIONS

DIAGNOSIS PROCEDURES

TEST MODULE PROCEDURES

SYSTEM VARIABLE INITIALIZATION & FIRST ENTRY POINT

CONTROL FOR CALLING ON TEST MODULE PROCEDURES

TERMINATE PROGRAM

FIGURE 6-2: LAYOUT OF THE ATLAS PROGRAM GENERATED BY THE CODE GENERATION

The above three stages are described in Sections 6.2 through 6.4. The following format is adhered to in describing the procedures in the above stages:

- Name of the procedure.
- Description of what it does.
- Description of data structures passed to it as external variables in PL/I.
- The algorithm, namely how the procedure accomplishes what it is supposed to do.

There are a number of support routines which are used jointly by all of the above three stages. They are described separately in Section 6.5. These support routines are contained as entry points in the procedures: ATLASIO and CONVERT The former contains routines which read and write into the files containing the ATLAS code. These routines output one line at a time to the ATLAS output files. They keep track of statement numbers in the ATLAS code and also allow forward GOTO statements. The procedure CONVERT contains routines for doing miscellaneous tasks such as character to number conversion; NOPAL identifier name to ATLAS identifier name conversion, etc.

6.2 Declarations and Diagnosis Generation Stage (Stage 1):

In this stage, declarations in ATLAS are generated for the user defined global variables, system variables (see Figure 6.3) and UUT connecting points. System variables are used to hold temporary results, to pass result of test modules, to keep status for each diagnosis and test module etc. Also procedures are generated corresponding to each one of the diagnoses in the NOPAL specification.

Four procedures are used to do all of the above: CDEMAIN, INITIAL, POSTDIAG and GENMSG. CDEMAIN generates the beginning of the ATLAS program and declarations of the user defined global variables. It calls INITIAL to generate declarations for system variables, connection points etc. Then for each of the diagnoses, it calls POSTDIAG (which in turn calls GENMSG) to generate procedure definition for each diagnosis. The above procedures are described in greater detail below.

In generating the system variables and variables corresponding to the user defined variables, diagnosis and UUT point names we suffix them with '.' followed by the name of the module. This is done to generate unique names in the programs generated for different modules becasue later on they will simply be put together to generate the total Atlas program. This became necessary because the Equate Atlas System does not have linker.

The generated diagnosis procedure has the following function to perform: It first decrements a counter which initially contains the number of test modules in conjunction for that diagnosis. If the value of the counter becomes less than 1, the actual diagnosis is performed. This normally consists of sending a message on operator console and waiting for appropriate response.

6.2.1 CDEMAIN:

The function of CDEMAIN has already been described above. We describe how it is accomplished below, in Algorithm 6.1.

Data structures to CEDMAIN (as EXTERNAL variables in PL/1):

The associative memory.

W(*,*) -- adjacency matrix from EXTSEQ.

ORDER(*) order vector from EXTSEQ.

RANK(*) rank vector from EXTSEQ

TEST-PTR(*) pointer array whose entries point to the storage entries of the diagnoses.

GL_VAR(*) structure array containing information about all the global variables.

#TESTS size of TEST_PTR array.

#DIAGS size of DIAG_PTR array

#VARS size of GL_VAR structure array.

#SIZE size of W matrix (Equal to the sum of #TESTS, #DIAGS and #VARS)

SPECNAME contains the name of the module.

ITISMOD True if it is not the main module.

There is a correspondence between entries in TEST_PTR, DIAG_PTR and GL_VAR arrays on one hand, and the entries in adjacency matrix W(*,*) on the other:

TEST_PTR(i) corresponds to ith entry in W.

DIAG_PTR(i) corresponds to $(\#TEST+i)^{th}$ entry in W.

GL_VAR(i) corresponds to $(\#TESTS+DIAGS+i)^{th}$ in W.

Data structures produced by CEDMAIN:

TSPECNAME contains the name of the module after it has been passed through CONVID. (CONVID makes it suitable for ATLAS).

DIAGNAME(*) produced by INITIAL. I^{th} entry contains the name of the I^{th} diagnosis (corresponding to DIAG_PTR(*)).

ALGORITHM 6.1: CDEMAIN:

1: Generate header for the ATLAS program (namely, 'BEGIN EQUATE PROGRAM<module name>') if ITISMOD is false.

2: Call INITIAL to generate declarations of system variables etc.

3: For each of the variables given by each of the entries in GL_VAR(*) do the following

i) Convert NOPAL identifier to ATLAS identifier by using proc CONVID (described later).

ii) Get its subscripts if any.

iii) Generate the ATLAS declaration.

For qualified variables (i.e. Prefix, '.'. suffix) generate declaration for suffix with an additional dimension in the declaration.

4: Define system utility routines for reading time from the clock in EQUATE-ATLAS.

5: Resolve the ATE functions referenced in the NOPAL specification and copy from the library FUNCLIB into the ATLASPROC file.

6: For each of the diagnosis given by entries in the DIAG-PTR(*) Issue a call to procedure POSTDIAG. (This procedure generates an ATLAS procedure for the desired diagnosis).

7: Return.

VARIABLE	TYPE	REMARKS
'SYS.#TESTS'	constant	It is set equal to the number of test modules.
'SYS.#DIAGS'	constant	It is set equal to the member of diagnoses.
'SYS.SELECTED' 'SYS.NOT SELECTED'	constant constant	Flags. Will be used to denote the status of diagnoses.
'SYS.TESTED' 'SYS.NOT TESTED' 'SYS.SKIPPED'		Flags. Will be used to denote the status of test modules.
'SYS.DIAG-FLAG'(I)	digital	Denotes whether the Ith diagnosis has been selected or not. I takes values from I to 'SYS.#DIAGS'.
'SYS.TEST FLAG'(I)	digital	Denotes whether the Ith test has been tested, not tested or skipped. I takes values from 1 to 'SYS.#TESTS'
'SYS.ASRT-FLAG'	digital	Denotes whether the outcome of the last assertion is true.
'SYS.FLAG'	digital	Denotes whether the outcome of the last test is true or false. It is used for selecting the diagnosis.
'SYS.Y/N'	digital	Used for storing the operator response in a diagnosis.
'SYS.#TEST_IN_CONJ'(I)	decimal	It is initialized to hold the number of test modules in conjunction for the Ith diagnosis. During the execution it gets decremented each time the Ith diagnosis is selected. I takes values from 1 to 'SYS.#DIAGS'.

FIGURE 6.3: LIST OF SYSTEM VARIABLES USED BY THE OBJECT PROGRAM

VARIABLE	TYPE	REMARKS
'SYS.TIM'	decimal	Time (value in seconds) as read from system clock at the start of ATLAS program.
'SYS.S-TIME'	decimal	Everytime a test module execution is begun time (in seconds) is read from system clock and the value is stored in 'SYS.S-TIME'.
'SYS.DECxx	decimal	Used to hold temporary results. xx is a two digit number
'SYS.DIGxx'	digital	-same-
'SYS.CLOCK'(*)	decimal	Used for storing the value read from the system clock.
'SYS.TIME'	decimal	The value in 'SYS.CLOCK'(*) is converted to seconds and stored in 'SYS.TIME'.
'SYS.GEN'	decimal	For modules. It will be incremented each time a modfun is called. It produces a new value, which corresponds to a new variable of the abstract data type.

FIGURE 6.3 (continued)

6.2.2 INITIAL:

This procedure generates the declarations in ATLAS of system defined global variables and UUT points in the ATLAS object code. It also generates a table of test modules and diagnoses and some comments.

Data structures passed to INITIAL:

The associative memory, of course.

TEST_PTR(*)	described in CDEMAIN
DIAG_PTR(*)	described in CEDMAIN
#TESTS	described in CDEMAIN
#DIAGS	described in CDEMAIN
TSPECNAME	described in CDEMAIN

Data structures produced by INITIAL:

DIAGNAME(*)	described in CDEMAIN

6.2.3 POSTDIAG:

This procedure is called with argument I such that DIAG_PTR(I) points to the storage entry of a diagnosis; and it produces a procedure in ATLAS for the diagnosis.

Data structures passed to POSTDIAG:

The associative memory.

DIAG_PTR(*)	described in CDEMAIN
DIAG_NAME(*)	described in CDEMAIN.
#DIAGS	described in CDEMAIN

ALGORITHM 6.2: INITIAL:

1: Generate a table of test module names (as comments in ATLAS) in the following way:

```
DO  I=1 TO #TESTS;
    Print name of the test given by TEST_PTR(I) in
    column 1 of the table.
    Print I in column 2 of the table.
    go to next row of the table.
end;
```

(Note that the number given to a particular test module will be used to refer to the test module in the rest of the code generation)

2: Generate a table of diagnosis names (as comments in ATLAS) as follows:

```
DO I = 1 to #DIAGS;
   Print name of the diagnosis given by DIAG_PTR(I)
   in column 1 of the table.
   Print I in column 2 of the table.
   Go to next row of the table.
end;
```

(The number assigned to a diagnosis will be used to refer to it in the rest of code generation)

3: Generate declarations of system variables.

i) Generate declaration for an array of flags SYS.DIAG_FLAG of dimension #DIAGS. (SYS.DIAG-FLAG(i) will refer to the diagnosis to which the number i has been assigned in Step 2 of this algorithm. Each of these

ALGORITHM 6.2 (continued)

flags will be used to denote whether the diagnosis has previously been selected).

ii) Generate declaration for an array of integers 'SYS.#TESTS IN CONJ' of dimension #DIAGS. (The ith element will store the number of tests which are in conjunction for selecting the i^{th} diagnosis. This gets decremented as the diagnosis is selected, and finally when it reaches 0 the diagnosis is actually issued).

iii) Generate declaration for the flag SYS.FLAG. (This flag is set to true or false depending on the outcome of the test).

iv) Miscellaneous other declarations.

4: For each of the UUT points do the following:

i) Retrieve the name of the UUT point

ii) If it has an alias

then retrieve Equate pin name using alias.

generate ('DEFINE' <UUT pin name><Equate pin name>)

else generate a comment with <UUT pin name>.

5: Define the routines for reading time from system clock.

6: Return.

Data structures produced by POSTDIAGS for use by GENMSG:

SYSCOMP(I)	Array of character strings containing all the failure functions and component ids occurring in I^{th} diagnosis.
SYSPARM(I)	Array of character strings containing the message parameters occurring in the I^{th} diagnosis.
PARMTYPE(I)	Array of numbers. PARMTYPE(i) e.g. whether it is of type integer, identifier, etc.
SYSOP	Char(3) var. Tells whether the diagnosis is selected by conjunction or disjunction of test modules.
NO_COMPS	size of SYSCOMP(*) array,
NO_PARMS	size of SYSPARM(*) array,

6.2.4 GENMSG

GENMSG is passed the message-number of a message which is to be printed in the ATLAS program. GENMSG retrieves the message, inserts components and parameters into the message (if necessary) and outputs the result to ATLAS.

Data structures passed to GENMSG:

The associative memory

SYSCOMP	described in POSTDIAG
SYSOP	described in POSTDIAG
SYSPARM	described in POSTDIAG
PARMTYPE	described in POSTDIAG
NO_COMPS	described in POSTDIAG
NO_PARMS	described in POSTDIAG

ALGORITHM 6.3: POSTDIAG(I):

(Storage entry of the diagnosis is pointed at by DIAG_PTR(I)).

1: Generate header for the diagnosis procedure.

2: Generate code to decrement 'SYS.#TESTS IN CONJ'. and if it is greater than Ø then return without executing the diagnosis procedure, if ITISMOD is false. (Purpose of this is to take care of those diagnosis which are to be selected in conjunction of test modules)

3: Retrieve all the affected components from associative memory and put them in SYSCOMP(*).

4: Retrieve all the message parameters and put them in SYSPARM(*) along with their type in PARMTYPE(*). If any SYSPARM(i) is a variable then call CONVID to convert it into a valid ATLAS identifier.

5: If timing value = 0 then;

generate read clock and compare with timing value.

generate delay if timing value is greater.

end if;

Call GENMSG; (GENMSG generates code which will issue a message at runtime. GENMSG does the substitutions for affected components and parameters in the message).

6: If the diagnosis has variables whose values are entered at execution time of diagnosis then generate ATLAS statements for the purpose.

(e.g. if 'A' gets its value at runtime of diagnosis generate ('INPUT "A"')

ALGORITHM 6.3: (continued)

7: If the diagnosis has Y_N = '?' i.e. it expects the operator to respond with YES/NO then generate the following:

'WAIT-FOR-MANUAL-INTERVENTION $'

8: Generate atlas code for setting the diagnosis flag to selected and end of diagnosis procedure:

'SYS.DIAG-FLAG'(i) = 'SYS.SELECTED' $.

END <diagname> $

9: Return.

ALGORITHM 6.4: GENMSG

Initialize local variables, retrieve the storage entry for the message, allocate text to hold the initial message, and allocate buffer to hold the final message (to be output)

```
1:   (Retrieve all message storage entries)
     DO I = 1 to # of entries
        IF correct storage entry (storage entry index = message#)
        then do;
          (allocate buffer, text)
           Length = Length of message + length components
           + length parameters allocate buffer, text
           fill text with message
     end;
     (above process is done in subroutines INIT and ALL_TX)
2:   (Replace QUOTES and double quotes with slashes)
        IF character = ' OR " then do;
           IF CHAR = ", replace with
           IF CHAR = 1, IF there follows a sequence of '#
           OR '.' then
              skip over these characters.  Otherwise replace
              with (above procedure uses subroutine chag-quote)
3:   (a '$C', '$P', or '$T' indicates that parameters or
     components are to be inserted into the buffer)
       IF character = '$' and next character = 'T' OR 'P' OR 'C'
       then do;
```

ALGORITHM 6.4: (continued):

```
        (FIND the number of components (if '$C') or
        the number of the parameter (if '$T' or '$P'))
        number of components = no_comps
        If $P OR $T then number of parameter = number of
        parameter + 1
        (Output the components or parameters to buffer)
     IF <component> ($C) then
        buffer=buffer||component||operator||components||op...
     IF parameter ($P) then
        buffer = buffer || parameter
     Goto   3
    END;
    ELSE GOTO 2
        (above procedure calls load-buf and OUTPUT_COMP)
4:  (Output the remaining test to the buffer and to ATLAS)
     buffer = buffer + remaining text.
     DO I = 1 to length of buffer by 60;
        EMIT TO ATLAS next 60 characters of buffer
        each atlas line begins with "and ends with", (OR
        endswith" if last line)
     Procedure GETNEXT CHAR returns the next character of
     the message text.
```

6.3 Intra-test code generation (Stage 2):

In this stage a procedure definition is generated for each of the test modules. The procedure CDETEST is called by INTSEQ which generates the procedure definition.

CDETEST generates code for conjunctions, assertions and calls to diagnosis procedures. Code for conjunctions and assertions is generated by calling GENWAVE with the appropriate argument. The procedure GENWAVE contains a number of procedures which are called depending on whether the waveform is a conjunction or an assertion, and whether it has an if-clause or is simple. Finally, GENEXPR is called which generates the actual code. All these are described in the following procedures:

6.3.1 CDETEST:

CDETEST is called with a pointer pointing to the storage entry of a test module as its argument. CDETEST produces a procedure in ATLAS corresponding to the test module. For modules, this procedure is called for the modfun which contains this test.

Data structures passed to CDETEST:

The associative memory.

(The following data structures contain data about the test module to which the argument of the procedure refers)

STIM_PTR(*)	Array of pointers which point to the storage entries of the stimuli waveforms in the test module.
MEAS_PTR(*)	Array of pointers which point to the storage entries of the measurement waveforms in the test module.

D_PTR(*)	Array of pointers which point to the storage entries of the diagnoses which are referenced by this test module.
LOC_VAR(*)	Array of structures which contain data about the variables (local and global) referenced by this test module.
FSUB(*)	structure array containing(i) name of variable used in loop (ii) upper limit of the loop. (Lower limit is always 1).
lDOTAB(*)	Structure array containing:
2 FIRST -->	index into order vector corresponding to beginning of loop.
2 LAST -->	index into order vector corresponding to end of loop.
2 SUB_# -->	index into FSUB which contains other information about this loop.
DIAGNAME(*)	described in CDEMAIN
#STIM	size of STIM_PTR(*)
#MEAS	size of MEAS_PTR(*)
#D	size of D_PTR(*)
LCLVAR	size of LOC_VAR(*) = #V
#W	= #STIM + #MEAS i.e. # of waveforms in the test-module.
#H	Number of nodes which are ancestors of nodes in LCLVAR.
SIZE	= #STIM+#MEAS+#D+#V+#H
ORDER(*)	Order vector for the adjacency matrix of the test module.

ALGORITHM 6.5: CDETEST(TEST):

(TEST is a pointer pointing to the storage entry of the test module)

1: DO I = 1 TO #D;

DIAGNO(I):= DIAGSEARCH(D_PTR(I));

END;

(Procedure DIAGSEARCH(P) searches for P in DIAG_PTR(*) and returns the index of the match. This establishes correspondence between the diagnoses in the test module and the number given to all the diagnoses by CDEMAIN)

2: TESTNO = TESTSEARCH(TEST);

(Procedure TESTSEARCH for test modules is similar to the procedure DIAGSEARCH for diagnoses)

3: Generate header in ATLAS for the testmodule procedure. If it is not the main module then for each of the formal source parameters of the corresponding modfun generate the following:

3.1 For the ith parameter name <pi> if it is source generate

< pi > = < test name>. PRMi

/* This parameter passing mechanism is described in Section 6.3.3. */

3.2 If it returns a variable S of data type same as the spec-name do the following:

3.2.1 Generate increment for the variable

'SYS.GEN ' spec name

3.2.1 Generate assignment

'S' = 'SYSGEN.spec-name

4: For each of the local variables in LOC_VAR(*) -
(it contains both local and global variables used by the test module) do the following:

i) Convert NOPAL id to ATLAS id (using CONVID).

ii) Suffix the id with TESTNO.

iii) Get its dimensions if it is an array.

iv) Generate the ATLAS declaration. Upperbound is taken to be 2 for variables with '*' or indefinite upper bound).

5: (Issue 0 time diagnosis)
DO I=1 TO #D;
If operator = '*' and timing = 0 for the diagnosis D_PTR(I) then generate call on the diagnosis procedure
END.

ALGORITHM 6.5: (continued)

```
6:   (Generate Code for Waveforms). Also check for iterations,
      if indicated by the structure DOTAB).
   DO J=1 TO SZ
      DO I=1 TO #LOOPS
         If ORDER(I) is a waveform then Call GENWAVE(pointer
                                                  to the waveform);
         If J = DOTAB(ENDAR(I)).LAST then generate end of loop;
         If J = DOTAB(ENDAR(I)).FIRST then generate beginning of
                                                  loop.

               If ORDER(I) is a source (or target) structure
               generate calls on ACCESS (or SAVE) functions.

      end;
  end;
7:   (Generate code for nonzero timing diagnosis preceded by
      proper logic.)
   DO for all diagnosis in the test module, with nonzero timing;
      If logic code = '|' or '&' then store diagnosis # in
                                                  TRUE_ARRAY;
      If logic code ='¦ ' or '& ' then store diagnosis # in
                                                  FALSE_ARRAY.
      If logic code = '*' then store diagnosis # in both
                                          TRUE_ARRAY and FALSE_ARRAY.
```

```
END;
```

8: Generate test for 'SYS.FLAG' equal to true.

Generate jump to true part and false part.

Generate calls on all the diagnosis in TRUE_ARRAY and put them in true part.

Generate calls on all the diagnosis in FALSE_ARRAY and put them in the false part.

There is a correspondence between entries in STIM_PTR, MEAS_PTR, D_PTR and LOC_VAR arrays on one hand and the entries in the adjacency matrix of the test module on the other.

STIM_PTR(i)	corresponds to i^{th} entry in the adjacency matrix
MEAS_PTR(i)	corresponds to $(\#STIM+i)^{th}$ in the adjacency matrix.
D_PTR(i)	corresponds to $(\#W+i)^{th}$ in the adjacency matrix.
LOC_VAR(i)	corresponds to $(\#W+\#D+i)^{th}$ in the adjacency matrix.

Data structures produced by CDETEST for use by GENWAVE

DIAGNO(*)	Array of integers of size #D. I^{th} entry in it gives the number assigned for code generation purposes to the diagnosis entry D_PTR(I). (For details about the number assigned to a diagnosis see step2 of CDEMAIN algorithm).

6.3.2 GENWAVE:

This procedure is called with a pointer pointing to storage entry of a waveform. It generates ATLAS code for the waveform.

Data structures passed to GENWAVE as external variables:

The associative memory.

ALGORITHM 6.6: GENWAVE (STOWAVE):

(STOWAVE points to storage entry of a waveform).

1. If the waveform is of type IF_CELL then call GENIFCELL
   else call GENSIMPLE

2. Return.

ALGORITHM 6.7: GENIFCELL(CELLPTR):

1: Generate ATLAS code to evaluate the condition (by calling GENEXPR(condition))

2: Generate compare statement followed by transfer of control to true part or false part depending on the outcome of the compare statement.

3: Generate code for true part by calling GENSIMPLE(true part);

4: Generate code for false part by calling GENIFCELL(false part) or GENSIMPLE(false part) as the case may be.

5: Return.

ALGORITHM 6.8: GENSIMPLE(WAVEPTR):

1. If waveform is of type assertion then do the following:

   (Assertions are in general case of the form,

   expr1 op expr2 +- range)

   i) T1 = GENEXPR(expr1)

   ii) T2 = GENEXPR(expr2)

   iii) If range = null then generate code T1 op T2

   else T3 = GENEXPR(range)

   generate compare T1 Upper limit T2+T3

   Lower limit T2-T3

   $

2. Else if waveform is of type conjunction then do the following:

   (conjunctions are of the form: <connection points>

   = <meas stim fn>)

   i) Generate code for each of the connection points by Calling GENEXPR(connection point) for each connection point.

   ii) Generate code for the measurement stimuli function by calling GENEXPR(<meas.stim.fn>)

3. Return;

6.3.3 Description of GENEXPR:

This is the heart of the GENWAVE procedure. Its argument EXPRTREE points to a parsed tree of either arithmetic expressions or boolean term or<connection dim. expression> or<func.dim.expr.>

It generates ATLAS code for the tree of an expression and returns one of two things:

1) the name of the variable to which the generated ATLAS code assigns the values of the expression,

or 2) the actual code itself.

In case (2) the code returned has not yet been issued to the object code file, and the purpose is to avoid low level inefficiency.

Consider the following example:

Expression is: SIN(A + Userfun(B,C))

GENEXPR is a recursive procedure. The following code will be generated on the recursive call

GENEXPR(Userfun(B,C)):

```
'Userfun.PRM01' = 'B' $
'Userfun.PRM02' = 'C' $
 PERFORM 'Userfun' $
 'SYS.DEC xx' = 'Userfun.RES'
```

(Note that the above mechanism of passing parameters to and returning result from user defined functions became necessary as ATLAS does not support passing the parameters and returning the result.)

This recursive call will return the variable 'SYS.TEMPxx', because the variable will contain the value of Userfun(B,C) at execution time.

Finally, the top level of call to GENEXPR will return the actual piece of code:

SIN('A' + 'SYS.DECxx').

In other words GENEXPR issues code into object file only when necessary and this way avoids unnecessary assignments.

(Note that if it issues code and returns variables only, it would also generate:

'SYS.DEC xy' = 'A' $

'SYS.DECxz' = 'SYS.DEC xy' + 'SYS. DECxx'$

'SYS.DEC zz' = SIN('SYS. DEC xz')$

and would finally return 'SYS. DEC zz'.)

Flag $NAMESW which is an external variable, is set if the value being returned by GENEXPR is of type variable; and is reset if the actual code is being returned.

ALGORITHM 6.9: GENEXPR(EXPRTREE):

1: If EXPRTREE is null then return ('**NULL**');

2: If EXPRTREE points to a number N then return NAMESW = '0'B; (count to char(N));

3: If EXPRTREE points to a character or bit string then NAMESW = '0'B; then return the string.

4: If EXPRTREE points to a tree such that the root FNNAME is user defined function or predicate with #ARGS as number of arguments, then do the following:

i) DO I = 1 TO #ARGS;

PARM(I) = GENEXPR(ARG(I));

PARMSW(I) = NAMESW;

PARMNAME(I) = FNNAME||'.PRM'||I;

Generate the assignment (PARMNAME(I)|| '=' ||PARM(I));

ii) Generate call ('PERFORM '|| FNNAME);

iii) Generate assignments to copy back the variable parameters, because we assume call by reference.

DO I = 1 TO #ARGS;

If PARMSW(I) then generate(PARM(I)||'='||PARMNAME(I));

END;

iv) Generate assignment to copy the result of the above function call in a temporary variable and return the temporary variable. Set NAMESW = 'Ø'B;

5. If EXPRTREE points to a variable or array or system fn. then do the following:

ALGORITHM 6.9: (continued)

Let FNNAME be the name of the variable array or system fn. and let #ARGS be its number of arguments. If FNNAME is a qualified variable (i.e. has a PREFIX, '.' and SUFFIX) then

```
    FNNAME = SUFFIX
    TEMPC = FNNAME||'('||PREFIX||','
else TEMPC = FNNAME||'('
TEMPC = TEMPC|| GENEXPR(ARG(1));
DO I=2 TO #ARGS;
    TEMPC = ','||GENEXPR(ARG(I));
END;
TEMPC = TEMPC||')';NAMESW='0'B;
Return(TEMPC);
```

6: If EXPRTREE points to a tree whose root is an operator: then do the following:

```
  NAMESW = '0'B;
  Return(GENEXPR(ARG(1))||operator||GENEXPR(ARG(2)));
```

7: If EXPRTREE points to value dim.expression then

```
  Return(GENEXPR(ARG(1)));
```

8: If EXPRTREE is none of the above Return ('**error**');

6.4 Inter Test Code Generation (Stage 3):

This is the final stage in which code is generated for calling the test module procedures in the right order preceded with appropriate logic. This stage is performed only for the main module.

The procedures used to do this job are TRMNATE, INVOKE and SKIP. The procedure TRMNATE calls INVOKE with the test module name as argument; the order of the test module names is given by the global order vector, which was generated by EXTSEQ. INVOKE generates logic which will determine at runtime whether the test module will be called.

TRMNATE and INVOKE are described in greater detail below:

6.4.1 TRMNATE:

This procedure generates the first entry point, i.e. the point from which the execution will begin, the control logic for calls on test module procedures by calling INVOKE, and finally termination of the ATLAS program.

Data structures passed to TRMNATE as external variables:

W(*,*)	described in CDEMAIN
ORDER(*)	described in CDEMAIN
RANK(*)	described in CDEMAIN
TEST-PTR(*)	described in CDEMAIN
DIAG_PTR(*)	described in CDEMAIN
#TESTS	described in CDEMAIN
#DIAGS	described in CDEMAIN
TSPECNAME	described in CDEMAIN
#SIZE	described in CDEMAIN

ALGORITHM 6.10: TRMNATE:

1: (1st entry point)

Generate ATLAS code to read time from system clock and print its value.

2: Generate code to initialize variables.

i) Initialize all 'SYS.DIAG-FLAG''s to NOT-SELECTED.

ii) Initialize 'SYS.#TESTS IN CONJ'(i) equal to the number of tests which are in conjunction for the diagnosis i, for i varying from 1 to #DIAGS.

iii) Initialize all 'SYS.TEST-FLAG''s to 'NOT TESTED'.

3: Generate the control logic for calls on test module procedures: The order given by the ORDER vector is used.

```
DO I=1 TO #SIZE;
   Call INVOKE(ORDER(I));
END;
```

4: Generate code to Read clock and print time.

Generate termination for ATLAS program.

6.4.2 INVOKE:

Generates control for calling the test module procedure given by the argument of call on INVOKE.

Data structures used by INVOKE (as external variables in PL/I

The associative memory

W(*,*)	Adjacency Matrix. Described in CDEMAIN.
TEST_PTR(*)	Described in CDEMAIN
#TESTS	Described in CDEMAIN
#DIAGS	Described in CDEMAIN
#SIZE	Described in CDEMAIN

6.4.3 SKIP(TESTNO, FLAG, STATE):

This procedure generates ATLAS code which will test FLAG '=' STATE and will mark the successors of TESTNO as skipped if true.

ALGORITHM 6.11: INVOKE(TESTNO):

(TESTNO is an integer referring to the test module which has been assigned the number TESTNO by the code generator. see CDEMAIN for details).

(The algorithm is based on negative logic. It generates ATLAS code for checking that all the predecessors of the test module given by TESTNO have been executed. If the answer will be true the test module procedure will be called. On the other hand if the answer will be false all the successors of the test modules will be marked as skipped. Atlas code is generated to take care of both of the above possibilities).

```
1:  Generate code to set the 'SYS.FLAG' to true.
2:  DO K=1 TO #DIAGS;
      PTYP = W(K+#TESTS,TESTNO);
      (PTYP now contains the precedence relationship between
      K^th diagnosis and the test module TESTNO.)
      If PTYP=2 OR PTYP=5 (The test module is connected to
              diagnosis by "after" or is such that its
              affected components set is a subset of D's)
      then generate ('SYS.FLAG' = 'SYS.FLAG' AND NOT
                          ('SYS.DIAG.FLAG'(K) XOR 'SYS.SELECTED'));
              Call SKIP(TESTNO, 'SYS.DIAG-FLAG'(K),'SYS.NOT-
                                                  SELECTED));
      Else if PTYP = 0 then
              generate('SYS.FLAG' = 'SYS.FLAG' AND NOT
                        ('SYS.DIAG-FLAG'(K) XOR'SYS.NOT SELECTED'));
              Call SKIP(TESTNO, 'SYS.DIAG-FLAG'(K),'SYS.SELECTED');
```

ALGORITHM 6.11: (continued)

3: Generate code to test 'SYS.FLAG' and call the test module procedure TESTNO if 'SYS.FLAG' is true.

4. Return.

ALGORITHM 6.12: SKIP:

```
1:  Generate code to test that 'SYS.TEST-FLAG'(TESTNO)
    has not been marked as 'SYS.SKIPPED'.
2:  Generate code to test that FLAG is equal to STATE.
3:  If any of the above return true then generate a
        GOTO to the end of the code generated by
        Step 2 of INVOKE.
4:  DO J = (#TESTS + #DIAGS + 1) TO #SIZE;
       If W(TESTNO,J)    =0 & EXAC_ONE_ENTRY(J) then
       (I is a target variable in test TESTNO and no other)
           DO I = 1 TO #TESTS;
            If W(J,I)-=0 then call SKIP(I,FLAG,STATE).
                (*Generate code to skip all the successors
                 of TESTNO=)
           END;
    END;
5:  Return.
```

6.5 The Support Routines:

There are two procedures ATLASIO and CONVERSION each of which contain a number of support routines. These support routines are contained as entry points, and are used by all the three stages of code generation.

6.5.1 Overview of ATLASIO :

This proc contains the entry points for writing into the files ATLAS, ATLASDCL and ATLASPROC. EMITDCL and COMENITDCL write into ATLASDCL and the others write into ATLASPROC. ATLASDCL contains only the DCL entries while ATLASPROC contains atlas proc and executable code.

6.5.1.1 Entry points

EMITDCL(X): This procedure prefixed the character string X with 9 blanks and writes it into the file ATLASDCL.

COMEMITDCL(X): Writes character string X, without prefixing into ATLASDCL.

EMITCOM(X): Writes character string X into ATLASPROC.

SAVELBL: It prefixes character string X with ATLSTMT, current statement number of ATLAS code, and stores it in the next available slot in an array of character strings called BUFFER. If BUFFER is already full then calls INCRIZ to increase the BUFFER size. This BUFFER will be emptied into ATLASPROC on call to DUMPSAV.

SAVE(X): This normally does the same thing as SAVEBL except that it prefixes X with blanks rather than ATLSTMT and also it does not increment ATLSTMT.

However if DMPFLAG is set then it does exactly the same thing as SAVELBL. (DMPFLAG gets set whenever the next line must have ATLSTMT as its label. This becomes necessary when a DUMPSAV has been performed and the forward 'goto' references have been resolved.)

EMIT(X): Normally it writes character string X into ATLASPROC after prefixing it with 9 blanks. However, if BUFFER is already allocated it calls SAVE(X). If DMPFLAG is set then it calls EMITLBL(X).

EMITBL(X): Normally it writes character string X into ATLASPROC, after ATLASPROC, after prefixing X with ATLSTMT. It increments ATLSTMT. However if BUFFER is allocated it calls SAVELBL(X). (BUFFER allocated implies that some previously issued character string have not been written to ATLASPROC and therefore character string X must be put in BUFFER to maintain the same sequence of character strings.)

INCRSIZE: Increases the size of array BUFFER from SIZE to (SIZE+INCR).

DUMPSAV(LABEL): It replaces the occurent of character string LABEL in the BUFFER with the value of ATLSTMT. If no more labels are present in BUFFER (labels are of the form '$$XX$$') then it empties the BUFFER into ATLASPROC.

SETATLAS: If vales of ATLSTMT is not a multiple of 100 then it is set to the next (highest) multiple of 100.

MERGEATLAS: It copies file ATLASDCL and then ATLASPROC into ATLAS.

FUNCLIB: Copies user defined function from file FLIB into ATLASDCL.

CONVT(T): Converts the integer T to character string corresponding to T and returns the string.

6.5.2 OVERVIEW OF CONVERT

There are a number of routines for various support purposes enclosed in procedure CONVERSION. Most of them are for conversions like: float to character string, fixed binary to character string, character string to fixed binary representation etc.

CONVID, CONVDIG, CONVTST convert the NOPAL identifiers to proper identifiers in EQUATE-ATLAS. For example: '_' in NOPAL is converted to '-' in EQUATE-ATLAS.

DIAGSEARCH(TESTSEARCH) are given diagnosis (test module) storage entry and return the number assigned to the particular diagnosis (test module) by the code generator. They are used principally by CDETEST.

APPENDIX A: A COMPLETE SET OF REPORTS FOR THE MINIRADIO EXAMPLE

In order to facilitate the discussion of the various aspects of the NOPAL language and demonstrate its usage, a sample test specification is presented below. This specification, referred to as MINIRADIOSET, along with a variety of reports produced by the NOPAL processor is presented below.

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 interconnecting points are defined.

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


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

NOPAL SPECIFICATION MINIRADIOSET;

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


  TEST DC_INPUT;

    /* NULL STIMULI */

    MEASUREMENT SM_DC_INPUT(DC_INPUT);

      CONJUNCTION SM_W0001(SM_DC_INPUT):
          (<J24_B, J24_C>  = OHMMETER(MRES OHM ))
                  TARGET: MRES;

      ASSERTION SP_W0002(SM_DC_INPUT):
          MRES >  100
                  SOURCE: MRES;

    LOGIC SLOGIC0010(DC_INPUT): |~INP_SHORTED, *DISPLAY;

      DIAGNOSIS INP_SHORTED:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=INPUT_SHORT,
              TYPE=SHORTED_MSG;

      DIAGNOSIS DISPLAY:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(MRES, ' OHMS'),
              TYPE=TEXT;

  TEST AMPL;

    /* NULL STIMULI */

    MEASUREMENT SM_AMPL(AMPL);

      ASSERTION SP_W0001(SM_AMPL):
          V_SIN =  0.26 +- 0.06
                  SOURCE: V_SIN;
```

FIGURE A.1: MINIRADIOSET NOPAL SPECIFICATION

```
   LOGIC SLOGIC0010(AMPL): *SHOW_MEAS, |~FREQ_TOL;

     DIAGNOSIS SHOW_MEAS:
         OPERATOR MESSAGE:
             OTHER PARAMETERS=(V_SIN, "VRMS"),
             TYPE=TEXT;

     DIAGNOSIS FREQ_TOL:
         OPERATOR MESSAGE:
             AFFECTED COMPONENTS=AMPL_TOL(STD_5MHZ_FRE),
             OTHER PARAMETERS=("AMPLIFIER"),
             TYPE=FREQ_TOL_MSG;

TEST DISTORT_2W;

   STIMULI SS_DISTORT_2(DISTORT_2W);

      CONJUNCTION SS_W0001(SS_DISTORT_2):
          (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &
          (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

   MEASUREMENT SM_DISTORT_2(DISTORT_2W);

      CONJUNCTION SM_W0001(SM_DISTORT_2):
          (<J19_L, GND>  = DISTORTION(M_DISTORT % ,1 KHZ ))
                   TARGET: M_DISTORT;

      ASSERTION SM_W0002(SM_DISTORT_2):
          M_DISTORT <= 5
                   SOURCE: M_DISTORT;

   LOGIC SLOGIC0010(DISTORT_2W): *DISTORT_PRNT, |~HI_DISTORTIO;

     DIAGNOSIS DISTORT_PRNT:
         OPERATOR MESSAGE:
             OTHER PARAMETERS=(M_DISTORT, "%"),
             TYPE=TEXT;

     DIAGNOSIS HI_DISTORTIO:
         OPERATOR MESSAGE:
             AFFECTED COMPONENTS=DISTORT(AUDIO_2W),
             OTHER PARAMETERS=(" 2W", 5.0),
             TYPE=DISTORT_MSG;

TEST DISTORT_VOLT;

   STIMULI DCV_AMS(DISTORT_VOLT);

      CONJUNCTION SS_W0001(DCV_AMS):
          (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &
          (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

   MEASUREMENT SM_DISTORT_V(DISTORT_VOLT);
```

FIGURE A.1: (continued)

```
        CONJUNCTION $M_W0001($M_DISTORT_V):
            (<J19_A, GND>  = SINE_WAVE(VRMS VOLT ,*,0 SEC ))
                    TARGET: VRMS;

        ASSERTION A1($M_DISTORT_V):
            VRMS >= 2.2
                    SOURCE: VRMS;

        ASSERTION A2($M_DISTORT_V):
            VRMS <= 2.8
                    SOURCE: VRMS;

    LOGIC $LOGIC0010(DISTORT_VOLT): *WAIT, *VRMS_PRNT, |~VRMS_FAILED;

      DIAGNOSIS WAIT:
          OPERATOR MESSAGE:
              TYPE=TUNE_MSG,
              TIME= 0.00000E+00SEC,
          RESPONSE=?;

      DIAGNOSIS VRMS_PRNT:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(VRMS, ' VAC'),
              TYPE=TEXT;

      DIAGNOSIS VRMS_FAILED:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=REF_VOLT(AUDIO_10MW),
              TYPE=VRMS_MSG;

TEST FREQ;

    STIMULI DCV(FREQ);

        CONJUNCTION $S_W0001(DCV):
            (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT ));

    MEASUREMENT $M_FREQ(FREQ);

        CONJUNCTION $M_W0001($M_FREQ):
            (<J22, GND>  = SINE_WAVE(V_SIN VOLT ,FREQ HZ ,DELAY_TIME SEC ))
                    TARGET: FREQ, V_SIN
                    SOURCE: DELAY_TIME;

        ASSERTION $M_W0002($M_FREQ):
            IF DELAY_TIME=60 THEN
                FREQ =  5E+06 +- 60
            ELSE
                FREQ =  5E+06 +- 2.5
                    SOURCE: FREQ, DELAY_TIME;

    LOGIC $LOGIC0010(FREQ): *GET_DELAY_TI, |~FREQ_TOL_FAI, *FREQ_PRNT;
```

FIGURE A.1: (continued)

```
      DIAGNOSIS GET_DELAY_TI:
          OPERATOR MESSAGE:
              TYPE=WARMUP_MSG,
              TIME= 0.000000E+00SEC,
          RESPONSE=(DELAY_TIME);

      DIAGNOSIS FREQ_TOL_FAI:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=FREQ_TOL(STD_5MHZ_FRE),
              OTHER PARAMETERS=('FREQ'),
              TYPE=FREQ_TOL_MSG;

      DIAGNOSIS FREQ_PRNT:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(FREQ, ' HZ'),
              TYPE=TEXT;

TEST DISTORT_10MW;

   STIMULI $S_DISTORT_1(DISTORT_10MW);

      CONJUNCTION $S_W0001($S_DISTORT_1):
          (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &
          (<J16>  = SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

   MEASUREMENT $M_DISTORT_1(DISTORT_10MW);

      CONJUNCTION $M_W0001($M_DISTORT_1):
          (<J19_A, GND>  = DISTORTION(M_DISTORT % ,2 KHZ ))
                  TARGET: M_DISTORT;

      ASSERTION $M_W0002($M_DISTORT_1):
          M_DISTORT <= 3
                  SOURCE: M_DISTORT;

   LOGIC $LOGIC0010(DISTORT_10MW): *DISTORT_PRNT, |~AUDIO_DISTOR;

    /*** FOLLOWING DIAGNOSIS ALREADY DEFINED BEFORE:

      DIAGNOSIS DISTORT_PRNT:
          OPERATOR MESSAGE:
              OTHER PARAMETERS=(M_DISTORT, '%'),
              TYPE=TEXT;
                                                          ***/

      DIAGNOSIS AUDIO_DISTOR:
          OPERATOR MESSAGE:
              AFFECTED COMPONENTS=DISTORT(AUDIO_10MW),
              OTHER PARAMETERS=(' 10MW', 1.0),
              TYPE=DISTORT_MSG;

   /*************************************************/
   /*                                               */
   /*  MESSAGES                                     */
```

FIGURE A.1: (continued)

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


   MESSAGE SHORTED_MSG:
       TEXT='R/T DC INPUT SHORTED J24-B/J24-C ', 'AN/GRC-106 DEFECTIVE.  CHECK P
RINTOUTS FOR DEFECTS.', 'PRESS STOP.';

   MESSAGE TEXT:  ALIAS=DISPLAY,
       TEXT='(P1):   (P2) ';

   MESSAGE FREQ_TOL_MSG:
       TEXT='(C) DEFECTIVE.', '5.0 MHZ STD. OUT OF (P) TOLERANCE.';

   MESSAGE DISTORT_MSG:
       TEXT='(P1) AUDIO DISTORTION GREATER THAN (P2) PERCENT.';

   MESSAGE TUNE_MSG:
       TEXT='TUNE RECEIVER:  MC & KC CONTROLS TO 250000.', 'ADJUST AUDIO GAIN CO
NTROL FOR 2.2 TO 2.8 VAC', '(2.5 VAC NOMINAL).    PRESS  YES.';

   MESSAGE VRMS_MSG:
       TEXT='10 MW DISTORTION REFERENCE VOLTAGE FAILED.';

   MESSAGE WARMUP_MSG:
       TEXT='IF A 12 MINUTE WARMUP IS DESIRED, ENTER 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 00600: AUDIO_2W, FAILURE FUNCTION=DISTORT;

   /*****************************************************/
   /*                                                   */
   /*  UUT CONNECTION POINTS                            */
   /*                                                   */
   /*****************************************************/
```

FIGURE A.1: (continued)

A-7

```
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      : J16, CONNECTOR=(COAXIAL, ),
       LIMIT=(UVOLT,  1.00000E+02,  0.00000E+00, GND),
       COMMENTS=' COAXIAL CABLE';

UUT_POINT      : J19_L, LIMIT=(VOLT,  7.00000E+01,  0.00000E+00, GND);

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: OHMMETER, FUNCTION TYPE=M, #PINS= 2,
       PARAM_01=(X, T, LIMIT=(OHM,  1.00000E+03,  1.00000E+00));

FUNCTION   120: AMPL_TOL, FUNCTION TYPE=F,
       PARAM_01=(COMPONENT, S);

FUNCTION    10: PWR_SUPPLY, FUNCTION TYPE=S, #PINS= 2,
       PARAM_01=(X, S, LIMIT=(VOLT,  6.00000E+01,  0.00000E+00));

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    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));

FUNCTION   140: DISTORT, FUNCTION TYPE=F,
       PARAM_01=(COMPONENT, S);

FUNCTION    30: SINE_WAVE, 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='AMPL., FREQ., TIME DELYD';

FUNCTION   130: REF_VOLT, FUNCTION TYPE=F,
```

FIGURE A.1: (continued)

```
        PARAM_01=(COMPONENT, S);

  FUNCTION   110: FREQ_TOL, FUNCTION TYPE=F,
        PARAM_01=(COMPONENT, S);

    /*****************************************************/
    /*                                                   */
    /*  ATE CONNECTION POINTS                            */
    /*                                                   */
    /*****************************************************/


  ATE_POINT      : ATE_J24B, UUT_POINTS=(J24_B);

END MINIRADIOSET;
```

FIGURE A.1: (continued)

CROSS REFERENCE AND ATTRIBUTES REPORT

NAME	DEF NO.	ATTRIBUTES AND REFERENCES
AMPL	9	TEST LABEL 10 12
AMPL_TOL	78	ATE-FUNCTION ID, F 14 68
ATE_J24B	81	ATE-POINT ID
AUDIO_DISTOR	51	DIAGNOSIS LABEL 49
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
DC_INPUT	2	TEST LABEL 3 6
DCV	35	STIMULUS LABEL 36
DCV_AMS	24	STIMULUS LABEL 25
DELAY_TIME	41	VARIABLE ID 38 39
DISPLAY	8	DIAGNOSIS LABEL 6
DISPLAY	52	SYNONYM OF MESSAGE LABEL: TEXT
DISTORT	80	ATE-FUNCTION ID, F 22 51 70 71
DISTORT_MSG	58	MESSAGE LABEL 22 51
DISTORT_PRNT	50	DIAGNOSIS LABEL 21 49
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
FREQ	34	TEST LABEL 35 37 40
FREQ	38	VARIABLE ID 39 43
FREQ_PRNT	43	DIAGNOSIS LABEL 40
FREQ_TOL	14	DIAGNOSIS LABEL 12
FREQ_TOL	77	ATE-FUNCTION ID, F 42 67
FREQ_TOL_FAI	42	DIAGNOSIS LABEL 40

FIGURE A.2: MINIRADIOSET NAME - STATEMENT - ATTRIBUTE CROSS REFERENCE REPORT

```
FREQ_TOL_MSG        55      MESSAGE LABEL
                              14  42
GET_DELAY_TI        41      DIAGNOSIS LABEL
                              40
GND                 65      SYNONYM OF UUT-POINT ID: J24_C, J19_B
                              17  19  25  27  36  38  45  47  59  60  62  63  64
HI_DISTORTIO        22      DIAGNOSIS LABEL
                              21
INP_SHORTED          7      DIAGNOSIS LABEL
                               6
INPUT_SHORT         66      COMPONENT ID
                               7
J16                 62      UUT-POINT ID
                              17  25  45
J19_A               63      UUT-POINT ID
                              27  47
J19_B               65      UUT-POINT ID
J19_L               64      UUT-POINT ID
                              19
J22                 59      UUT-POINT ID
                              38
J24_B               60      UUT-POINT ID
                               4  17  25  36  45  81
J24_C               61      UUT-POINT ID
                               4
M_DISTORT           19      VARIABLE ID
                              20  50
M_DISTORT           47      VARIABLE ID
                              48  50
MINIRADIOSET         1      SPECIFICATION LABEL
                              82
MRES                 4      VARIABLE ID
                               5   8
OHMMETER            73      ATE-FUNCTION ID, M
                               4
PWR_SUPPLY          72      ATE-FUNCTION ID, S
                              17  25  36  45
REF_VOLT            79      ATE-FUNCTION ID, F
                              33  69
SAM                 76      SYNONYM OF ATE-FUNCTION ID: SIGNAL_AM, S
SHORTED_MSG         53      MESSAGE LABEL
                               7
SHOW_MEAS           13      DIAGNOSIS LABEL
                              12
SIGNAL_AM           76      ATE-FUNCTION ID, S
                              17  25  45
SINE_DELAY          74      SYNONYM OF ATE-FUNCTION ID: SINE_WAVE, M
SINE_WAVE           74      ATE-FUNCTION ID, M
                              27  38
STD_5MHZ_FRE        67      COMPONENT ID, WITH FAILURE-FUNCTION: FREQ_TOL
                              42
STD_5MHZ_FRE        68      COMPONENT ID, WITH FAILURE-FUNCTION: AMPL_TOL
                              14
TEXT                52      MESSAGE LABEL
                               8  13  32  43  50
TUNE_MSG            56      MESSAGE LABEL
```

FIGURE A.2: (continued)

```
                                31
V_SIN                   38      VARIABLE ID, GLOBAL
                                13  11
VRMS                    27      VARIABLE ID
                                28  29  32
VRMS_FAILED             33      DIAGNOSIS LABEL
                                30
VRMS_MSG                57      MESSAGE LABEL
                                33
VRMS_PRNT               32      DIAGNOSIS LABEL
                                30
WAIT                    31      DIAGNOSIS LABEL
                                30
WARMUP_MSG              54      MESSAGE LABEL
                                41
XJ24_B                  60      SYNONYM OF UUT-POINT ID: J24_B
1                       66      COMPONENT/FAILURE SEQ#
                                67  68  69  70
2                       67      COMPONENT/FAILURE SEQ#
3                       68      COMPONENT/FAILURE SEQ#
6                       69      COMPONENT/FAILURE SEQ#
                                70
7                       70      COMPONENT/FAILURE SEQ#
```

FIGURE A.2: (continued)

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

DIAGNOSIS	TEST-MODULES
INP_SHORTED	DC_INPUT
DISPLAY	DC_INPUT
SHOW_MEAS	AMPL
FREQ_TOL	AMPL
DISTORT_PRNT	DISTORT_2W, DISTORT_10MW
HI_DISTORTIO	DISTORT_2W
WAIT	DISTORT_VOLT
VRMS_PRNT	DISTORT_VOLT
VRMS_FAILED	DISTORT_VOLT
GET_DELAY_TI	FREQ
FREQ_TOL_FAI	FREQ
FREQ_PRNT	FREQ
AUDIO_DISTOR	DISTORT_10MW

FIGURE A.3: OTHER MINIRADIOSET CROSS REFERENCE REPORT

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

MESSAGE	DIAGNOSES	TEST-MODULES
SHORTED_MSG	INP_SHORTED	DC_INPUT
TEXT/DISPLAY	DISPLAY, SHOW_MEAS, VRMS_PRNT, FREQ_PRNT, DISTORT_PRNT	DC_INPUT, AMPL, DISTORT_VOLT, FREQ, DISTORT_2W, DISTORT_10RW
FREQ_TOL_MSG	FREQ_TOL, FREQ_TOL_FAI	AMPL, FREQ
DISTORT_MSG	HI_DISTORTIO, AUDIO_DISTOR	DISTORT_2W, DISTORT_10RW
TUNE_MSG	WAIT	DISTORT_VOLT
VRMS_MSG	VRMS_FAILED	DISTORT_VOLT
WARMUP_MSG	GET_DELAY_TI	FREQ

FIGURE A.3: (continued)

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

AFFECTED-COMPONENT	DIAGNOSES	TEST-MODULES
1: INPUT_SHORT	INP_SHORTED	DC_INPUT
2: FREQ_TOL(STD_5MHZ_FRE)	FREQ_TOL_FAI	FREQ
3: AMPL_TOL(STD_5MHZ_FRE)	FREQ_TOL	AMPL
6: REF_VOLT(AUDIO_10MW)	VRMS_FAILED	DISTORT_VOLT
7: DISTORT(AUDIO_10MW)	AUDIO_DISTOR	DISTORT_10MW
00600: DISTORT(AUDIO_2W)	HI_DISTORTIO	DISTORT_2W

FIGURE A.3: (continued)

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), DISTORT_2W(S), DISTORT_VOLT(S), FREQ(S), DISTORT_10MW(S)	ATE_J24B
J24_C/GND	DC_INPUT(M), DISTORT_2W(S), DISTORT_2W(M), DISTORT_VOLT(S), DISTORT_VOLT(M), FREQ(S), FREQ(M), DISTORT_10MW(S), DISTORT_10MW(M)	
J16	DISTORT_2W(S), DISTORT_VOLT(S), DISTORT_10MW(S)	
J19_L	DISTORT_2W(M)	
J19_A	DISTORT_VOLT(M), DISTORT_10MW(M)	
J22	FREQ(M)	
J19_B/GND	DISTORT_2W(S), DISTORT_2W(M), DISTORT_VOLT(S), DISTORT_VOLT(M), FREQ(S), FREQ(M), DISTORT_10MW(S), DISTORT_10MW(M)	

FIGURE A.3: (continued)

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

ATE-FUNCTION,TYPE	TEST-MODULES(S/M)
OHMMETER, M	DC_INPUT(M)
PWR_SUPPLY, S	DISTORT_2W(S), DISTORT_VOLT(S), FREQ(S), DISTORT_10MW(S)
SIGNAL_AM/SAM, S	DISTORT_2W(S), DISTORT_VOLT(S), DISTORT_10MW(S)
DISTORTION, M	DISTORT_2W(M), DISTORT_10MW(M)
SINE_WAVE/SINE_DELAY, M	DISTORT_VOLT(M), FREQ(M)

FIGURE A.3: (continued)

WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION MINIRADIOSET

page 1

			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	0	9	0	0	11	13	0	0	0	0	0	0	0	0	0	0
3	DISTORT_2W	TEST	0	0	0	0	0	0	0	0	0	0	11	13	0	0	0	0	0	0	0	0
4	DISTORT_VOLT	TEST	0	0	0	0	0	0	0	0	0	0	0	0	11	11	13	0	0	0	0	0
5	FREQ	TEST	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	13	11	0	1
6	DISTORT_10MW	TEST	0	0	0	0	0	0	0	0	0	0	11	0	0	0	0	0	0	0	13	0
7	INP_SHORTED	DIAGNOSIS	0	4	0	4	4	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	DISPLAY	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	SHOW_MEAS	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	FREQ_TOL	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	DISTORT_PRNT	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	HI_DISTORTIO	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13	WAIT	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	VRMS_PRNT	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15	VRMS_FAILED	DIAGNOSIS	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16	GET_DELAY_TI	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	FREQ_TOL_FAI	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	FREQ_PRNT	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
19	AUDIO_DISTOR	DIAGNOSIS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20	V_SIN	VARIABLE	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION MINIRADIOSET

page 2

ORDER VECTOR INDEX	ORDER VECTOR	RANK	TYPE	NAME
1	1	0	TEST	DC_INPUT
2	7	1	DIAGNOSIS	INP_SHORTED
3	8	1	DIAGNOSIS	DISPLAY
4	5	2	TEST	FREQ
5	16	3	DIAGNOSIS	GET_DELAY_TI
6	17	3	DIAGNOSIS	FREQ_TOL_FAI
7	18	3	DIAGNOSIS	FREQ_PRNT
8	20	3	GLOBAL VARIABLE	V_SIN
9	2	4	TEST	AMPL
10	3	5	TEST	DISTORT_2W
11	4	5	TEST	DISTORT_VOLT
12	9	5	DIAGNOSIS	SHOW_MEAS
13	10	5	DIAGNOSIS	FREQ_TOL
14	12	6	DIAGNOSIS	HI_DISTORTIO
15	13	6	DIAGNOSIS	WAIT
16	14	6	DIAGNOSIS	VRMS_PRNT
17	15	6	DIAGNOSIS	VRMS_FAILED
18	6	7	TEST	DISTORT_10MW
19	11	8	DIAGNOSIS	DISTORT_PRNT
20	19	8	DIAGNOSIS	AUDIO_DISTOR

FIGURE A.4: INTER-TEST AND INTRA-TEST ADJACENCY MATRICE AND ORDER VECTOR FOR MINIRADIOSET

```
                    INTRA MODULE SEQUENCING DC_INPUT
                    ANALYSIS OF THE ADJACENCY MATRIX
                                1 2 3 4 5
                                - - - - -
    1  SM_W0001      CONJUNCTION   0 0 0 0 1
    2  SM_W0002      ASSERTION     0 0 0 0 0
    3  INP_SHORTED   DIAGNOSES     0 0 0 0 0
    4  DISPLAY       DIAGNOSES     0 0 0 0 0
    5  MRES          VARIABLE      0 1 0 1 0

                    SEQUENCE OF PROCESSING FOR TEST DC_INPUT

ORDER
VECT  ORDER   RANK     NAME          TYPE          TEXT
INDEX VECTOR
  1     1     0      SM_W0001      CONJUNCTION   (<J24_B, J24_C>  = OHMMETER(MRES OHM ))
                                                 TARGET:  MRES;

  2     3     0      INP_SHORTED   DIAGNOSES     AFFECTED COMPONENTS = INPUT_SHORT,  TYPE = SHORTED_MSG;

  3     5     1      MRES          VARIABLE      LOCAL

  4     2     2      SM_W0002      ASSERTION     MRES >  100
                                                 SOURCE:  MRES;

  5     4     2      DISPLAY       DIAGNOSES     OTHER PARAMETERS=(' OHMS', MRES),  TYPE = TEXT;
```

FIGURE A.4: (continued)

```
                    INTRA MODULE SEQUENCING AMPL
                    ANALYSIS OF THE ADJACENCY MATRIX
                                1 2 3 4
                                - - - -
   1  SM_W0001      ASSERTION    0 0 0 0
   2  SHOW_MEAS     DIAGNOSES    0 0 0 0
   3  FREQ_TOL      DIAGNOSES    0 0 0 0
   4  V_SIN         VARIABLE     1 1 0 0

                    SEQUENCE OF PROCESSING FOR TEST AMPL

ORDER
VECT  ORDER   RANK     NAME          TYPE                    TEXT
INDEX VECTOR
  1     3     0      FREQ_TOL      DIAGNOSES      AFFECTED COMPONENTS = AMPL_TOL(STD_SMNZ_FRE), OTHER PARAMETERS=(
                                                  "AMPLIFIER"),  TYPE = FREQ_TOL_MSG;

  2     4     0      V_SIN         VARIABLE       GLOBAL / SOURCE /

  3     1     1      SM_W0001      ASSERTION      V_SIN =  0.26 +- 0.06
                                                  SOURCE:  V_SIN;

  4     2     1      SHOW_MEAS     DIAGNOSES      OTHER PARAMETERS=("VRMS", V_SIN),  TYPE = TEXT;
```

FIGURE A.4: (continued)

```
                    INTRA MODULE SEQUENCING DISTORT_2W
                    ANALYSIS OF THE ADJACENCY MATRIX
                                1 2 3 4 5 6
                                - - - - - -
   1  SS_W0001      CONJUNCTION   0 2 0 0 0 0
   2  SM_W0001      CONJUNCTION   0 0 0 0 0 1
   3  SM_W0002      ASSERTION     0 0 0 0 0 0
   4  DISTORT_PRNT  DIAGNOSES     0 0 0 0 0 0
   5  MI_DISTORTIO  DIAGNOSES     0 0 0 0 0 0
   6  M_DISTORT     VARIABLE      0 0 1 1 0 0

                    SEQUENCE OF PROCESSING FOR TEST DISTORT_2W
ORDER
VECT  ORDER   RANK    NAME          TYPE          TEXT
INDEX VECTOR
  1     1     0     SS_W0001      CONJUNCTION   (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &(<J16>  =
                                                 SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

  2     5     0     MI_DISTORTIO  DIAGNOSES     AFFECTED COMPONENTS = DISTORT(AUDIO_2W), OTHER PARAMETERS=(" 2W",
                                                 5.0),  TYPE = DISTORT_MSG;

  3     2     1     SM_W0001      CONJUNCTION   (<J19_L, GND>  = DISTORTION(M_DISTORT % ,1 KHZ ))
                                                 TARGET:  M_DISTORT;

  4     6     2     M_DISTORT     VARIABLE      LOCAL

  5     3     3     SM_W0002      ASSERTION     M_DISTORT <= 5
                                                 SOURCE:  M_DISTORT;

  6     4     3     DISTORT_PRNT  DIAGNOSES     OTHER PARAMETERS=("%", M_DISTORT),  TYPE = TEXT;
```

FIGURE A.4: (continued)

```
                    INTRA MODULE SEQUENCING DISTORT_VOLT
                    ANALYSIS OF THE ADJACENCY MATRIX
                                1 2 3 4 5 6 7 8
                                - - - - - - - -
  1  SS_#0001      CONJUNCTION   0 2 0 0 0 0 0 0
  2  SM_#0001      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  WAIT          DIAGNOSES     0 0 0 0 0 0 0 0
  6  VRMS_PRNT     DIAGNOSES     0 0 0 0 0 0 0 0
  7  VRMS_FAILED   DIAGNOSES     0 0 0 0 0 0 0 0
  8  VRMS          VARIABLE      0 0 1 1 0 1 0 0

                    SEQUENCE OF PROCESSING FOR TEST DISTORT_VOLT

ORDER
VECT  ORDER   RANK    NAME          TYPE          TEXT
INDEX VECTOR
  1     1     0     SS_#0001      CONJUNCTION   (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &(<J16>  =
                                                 SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

  2     5     0     WAIT          DIAGNOSES     TYPE = TUNE_MSG, TIME= 0.000000E+00,  RESPONSE = ?;

  3     7     0     VRMS_FAILED   DIAGNOSES     AFFECTED COMPONENTS = REF_VOLT(AUDIO_10MW),  TYPE = VRMS_MSG;

  4     2     1     SM_#0001      CONJUNCTION   (<J19_A, GND>  = SINE_WAVE(VRMS VOLT ,*,0 SEC ))
                                                 TARGET:  VRMS;

  5     8     2     VRMS          VARIABLE      LOCAL

  6     3     3     A1            ASSERTION     VRMS >= 2.2
                                                 SOURCE:  VRMS;

  7     4     3     A2            ASSERTION     VRMS <= 2.8
                                                 SOURCE:  VRMS;

  8     6     3     VRMS_PRNT     DIAGNOSES     OTHER PARAMETERS=(" VAC", VRMS),  TYPE = TEXT;
```

FIGURE A.4: (continued)

```
                    INTRA MODULE SEQUENCING FREQ
                    ANALYSIS OF THE ADJACENCY MATRIX
                              1 2 3 4 5 6 7 8 9
                              - - - - - - - - -
 1  SS_W0001       CONJUNCTION   0 2 0 0 0 0 0 0 0
 2  SM_W0001       CONJUNCTION   0 0 0 0 0 0 0 1 1
 3  SM_W0002       ASSERTION     0 0 0 0 0 0 0 0 0
 4  GET_DELAY_TI   DIAGNOSES     0 0 0 0 0 0 1 0 0
 5  FREQ_TOL_FAI   DIAGNOSES     0 0 0 0 0 0 0 0 0
 6  FREQ_PRNT      DIAGNOSES     0 0 0 0 0 0 0 0 0
 7  DELAY_TIME     VARIABLE      0 1 1 0 0 0 0 0 0
 8  FREQ           VARIABLE      0 0 1 0 0 1 0 0 0
 9  V_SIN          VARIABLE      0 0 0 0 0 0 0 0 0

                    SEQUENCE OF PROCESSING FOR TEST FREQ

ORDER
VECT  ORDER   RANK    NAME            TYPE                    TEXT
INDEX VECTOR
  1     1     0     SS_W0001       CONJUNCTION     (<J24_8, GND>  = PWR_SUPPLY(27.5 VOLT ));

  2     4     0     GET_DELAY_TI   DIAGNOSES       TYPE = WARMUP_MSG, TIME= 0.00000E+00,  RESPONSE=( DELAY_TIME);

  3     5     0     FREQ_TOL_FAI   DIAGNOSES       AFFECTED COMPONENTS = FREQ_TOL(STD_5MHZ_FRE), OTHER PARAMETERS=(
                                                   "FREQ"),  TYPE = FREQ_TOL_MSG;

  4     7     1     DELAY_TIME     VARIABLE        LOCAL

  5     2     2     SM_W0001       CONJUNCTION     (<J22, GND>  = SINE_WAVE(V_SIN VOLT ,FREQ HZ ,DELAY_TIME SEC ))
                                                   TARGET:  FREQ,  V_SIN
                                                   SOURCE:  DELAY_TIME;

  6     8     3     FREQ           VARIABLE        LOCAL

  7     9     3     V_SIN          VARIABLE        GLOBAL / TARGET /

  8     3     4     SM_W0002       ASSERTION       IF DELAY_TIME=60 THEN FREQ =  5E+06 +- 60
                                                   ELSE FREQ =  5E+06 +- 2.5
                                                   SOURCE:  DELAY_TIME,  FREQ;

  9     6     4     FREQ_PRNT      DIAGNOSES       OTHER PARAMETERS=(" HZ", FREQ),  TYPE = TEXT;
```

FIGURE A.4: (continued)

```
                    INTRA MODULE SEQUENCING DISTORT_10MW
                    ANALYSIS OF THE ADJACENCY MATRIX
                                1 2 3 4 5 6
                                - - - - - -
  1  SS_W0001       CONJUNCTION 0 2 0 0 0 0
  2  SM_W0001       CONJUNCTION 0 0 0 0 0 1
  3  SM_W0002       ASSERTION   0 0 0 0 0 0
  4  DISTORT_PRNT   DIAGNOSES   0 0 0 0 0 0
  5  AUDIO_DISTOR   DIAGNOSES   0 0 0 0 0 0
  6  M_DISTORT      VARIABLE    0 0 1 1 0 0

                    SEQUENCE OF PROCESSING FOR TEST DISTORT_10MW

ORDER
VECT  ORDER   RANK    NAME          TYPE          TEXT
INDEX VECTOR
   1     1     0     SS_W0001      CONJUNCTION   (<J24_B, GND>  = PWR_SUPPLY(27.5 VOLT )) &(<J16>  =
                                                  SIGNAL_AM(25.002 MHZ ,+13 DB ,0 % ,1 KHZ ));

   2     5     0     AUDIO_DISTOR  DIAGNOSES     AFFECTED COMPONENTS = DISTORT(AUDIO_10MW), OTHER PARAMETERS=("
                                                  10MW", 1.0),  TYPE = DISTORT_MSG;

   3     2     1     SM_W0001      CONJUNCTION   (<J19_A, GND>  = DISTORTION(M_DISTORT % ,2 KHZ ))
                                                  TARGET:  M_DISTORT;

   4     6     2     M_DISTORT     VARIABLE      LOCAL

   5     3     3     SM_W0002      ASSERTION     M_DISTORT <= 3
                                                  SOURCE:  M_DISTORT;

   6     4     3     DISTORT_PRNT  DIAGNOSES     OTHER PARAMETERS=("%", M_DISTORT),  TYPE = TEXT;
```

FIGURE A.4: (continued)

```
1
                              FLOWCHART REPORT FOR NOPAL SPECIFICATION MINIRADIOSET
0       /***** FLOWCHART FOR GLOBAL PROCEDURE MINIRADIOSET *****/
0         BEGIN MINIRADIOSET;
0               DECLARE AS GLOBAL VARIABLES:
                      V_SIN;
0               DECLARE AS EVAL OR CONTROL FUNCTIONS:
                      /* NO EVAL OR CONTROL FUNCTIONS */
0               DECLARE AS STIMULUS FUNCTIONS:
                      SIGNAL_AM, SAM, PWR_SUPPLY;
0               DECLARE AS MEASUREMENT FUNCTIONS:
                      SINE_WAVE, SINE_DELAY, DISTORTION, OHMMETER;
0               DECLARE AS FAILURE FUNCTIONS:
                      FREQ_TOL, REF_VOLT, DISTORT, AMPL_TOL;
0             /*  EXECUTE LEVEL     0 */
                PERFORM TEST DC_INPUT;
0             /*  EXECUTE LEVEL     2 */
                PERFORM TEST FREQ;
0             /*  EXECUTE LEVEL     4 */
                PERFORM TEST AMPL;
0             /*  EXECUTE LEVEL     5 */
                PERFORM TEST DISTORT_2W;
                PERFORM TEST DISTORT_VOLT;
0             /*  EXECUTE LEVEL     7 */
                PERFORM TEST DISTORT_10MW;
0         END MINIRADIOSET;
```

FIGURE A.5: MINIRADIOSET FLOWCHART REPORT

```
 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   21, NEAR TEXT "HI_DISTORTIO", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   22, NEAR TEXT "HI_DISTORTIO", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   40, NEAR TEXT "GET_DELAY_TI", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   40, NEAR TEXT "FREQ_TOL_FAI", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   41, NEAR TEXT "GET_DELAY_TI", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   42, NEAR TEXT "FREQ_TOL_FAI", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   42, NEAR TEXT "STD_5MHZ_FRE", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   49, NEAR TEXT "AUDIO_DISTOR", NAME/INTEGER WAS TOO LONG.  TRUNCATED.
*WARNING*   IN STATEMENT   51, NEAR TEXT "AUDIO_DISTOR", 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 =   12  , NO. OF STATEMENTS=   82


ERROR/WARNING MESSAGES GENERATED DURING CROSS-REFERENCE:
*STATISTICS*  NO. OF XREF1 ERRORS =    0     NO. OF WARNINGS =    0


ERROR/WARNING MESSAGES GENERATED DURING SEQUENCING:
 *STATISTICS* NO. OF SEQUENCING ERRORS=    0, NO. OF WARNINGS=    0
```

FIGURE A.6: ERROR AND WARNING REPORT FOR MINIRADIOSET

```
      BEGIN EQUATE PROGRAM 'MINIRADIOSET' $
C          *** TEST MODULES ***
C        TEST MODULE              ATLAS
C           NAMES               SUBSCRIPTS
C             'DC-INPUT'          -- 1
C             'AMPL'              -- 2
C             'DISTORT-2W'        -- 3
C             'DISTORT-VOLT'      -- 4
C             'FREQ'              -- 5
C             'DISTORT-10MW'      -- 6
C
C             *** DIAGNOSES ***
C        DIAGNOSES                ATLAS
C          NAMES                SUBSCRIPTS
C             'INP_SHORTED'       -- 1
C             'DISPLAY'           -- 2
C             'SHOW_MEAS'         -- 3
C             'FREQ_TOL'          -- 4
C             'DISTORT_PRNT'      -- 5
C             'HI_DISTORTIO'      -- 6
C             'WAIT'              -- 7
C             'VRMS_PRNT'         -- 8
C             'VRMS_FAILED'       -- 9
C             'GET_DELAY_TI'      -- 10
C             'FREQ_TOL_FAI'      -- 11
C             'FREQ_PRNT'         -- 12
C             'AUDIO_DISTOR'      -- 13
C     $
C    DECLARATIONS OF SYSTEM VARIABLES  $
        DECLARE DIGITAL, LIST,'SYS.DIAG-FLAG'(13) $
        DECLARE DECIMAL 'SYS.S-TIME' $
        DECLARE DECIMAL 'SYS.DUMMY' $
        DECLARE DECIMAL, LIST,'SYS.#TESTS IN CONJ'(13) $
        DECLARE DIGITAL, LIST, 'SYS.TEST-FLAG'(6) $
        DECLARE DIGITAL, 'SYS.FLAG' $
        DECLARE DECIMAL, 'SYS.TIM', 'SYS.TIME' $
        DECLARE DECIMAL, LIST, 'SYS.CLOCK'(6)  $
        DECLARE DECIMAL, 'SYS.I' $
        DECLARE DIGITAL, 'SYS.Y/N' $
C    *** CONSTANTS ***  $
        DEFINE 'SYS.SELECTED' , B'10' $
        DEFINE 'SYS.NOT SELECTED' , B'01' $
        DEFINE 'SYS.NOT TESTED' , B'00' $
        DEFINE 'SYS.TESTED' , B'10' $
        DEFINE 'SYS.SKIPPED' , B'01' $
        DEFINE 'SYS.#DIAGS' ,  13       $
        DEFINE 'SYS.#TESTS' ,  6        $
        DEFINE 'SYS.TRUE' , B'1' $
        DEFINE 'SYS.FALSE' , B'0' $
C    UUT POINT DEFINITIONS     $
        DEFINE 'J19-B', GND $
C    UUT PIN NAME NOT ASSIGNED: 'J22' $
C    UUT PIN NAME NOT ASSIGNED: 'J19-A' $
C    UUT PIN NAME NOT ASSIGNED: 'J19-L' $
```

FIGURE A.7: MINIRADIOSET ATLAS PROGRAM

```
C     UUT PIN NAME NOT ASSIGNED: 'J16' $
          DEFINE 'J24-C', GND $
          DEFINE 'J24-B', XJ24-B $
C     FOLLOWING DISK FILE SHOULD CONTAIN THE
C     MISSING EQUATE/UUT PIU & DIU ASSIGNMENTS. $
          INCLUDE "CNXMIN" $
C     MACRO DEFINITIONS $
          DEFINE 'PRT.TIME', 'SYS.CLOCK'(1), " ##:",
                 'SYS.CLOCK'(2), "##:", 'SYS.CLOCK'(3), "##" $
C     DECLARATIONS FOR USER DEFINED GLOBAL VARIABLES   $
          DECLARE DECIMAL,  'V-SIN'  $
C     SYSTEM UTILITY ROUTINES  $
          DEFINE PROCEDURE 'GET.TIME' $
            READ(TIME 'SYS.CLOCK'(1) ALL), SYS-CLOCK $
            'SYS.TIME' = 3600*'SYS.CLOCK'(1) + 60*'SYS.CLOCK'(2) +
                  'SYS.CLOCK'(3) $
          END 'GET.TIME' $
          DECLARE DECIMAL 'SYS.TEMP.01' $
          DECLARE DECIMAL 'SYS.TEMP.02' $
          DECLARE DECIMAL 'SYS.TEMP.03' $
          DECLARE DECIMAL 'SYS.TEMP.04' $
          DECLARE DECIMAL 'SYS.TEMP.05' $
          DECLARE DECIMAL 'SYS.TEMP.06' $
          DECLARE DECIMAL 'SYS.TEMP.07' $
          DECLARE DECIMAL 'SYS.TEMP.08' $
          DECLARE DECIMAL 'SYS.TEMP.09' $
          DECLARE DECIMAL 'SYS.TEMP.10' $
          DECLARE DECIMAL 'SYS.TEMP.11' $
          DECLARE DECIMAL 'SYS.TEMP.12' $
          DECLARE DECIMAL 'SYS.TEMP.13' $
          DECLARE DECIMAL 'SYS.TEMP.14' $
          DECLARE DECIMAL 'SYS.TEMP.15' $
          DECLARE DECIMAL 'SYS.TEMP.16' $
          DECLARE DECIMAL 'SYS.TEMP.17' $
          DECLARE DECIMAL 'SYS.TEMP.18' $
          DECLARE DECIMAL 'SYS.TEMP.19' $
          DECLARE DECIMAL 'SYS.TEMP.20' $
          DECLARE DECIMAL 'SYS.TEMP.21' $
          DECLARE DECIMAL 'SYS.TEMP.22' $
          DECLARE DECIMAL 'SYS.TEMP.23' $
          DECLARE DECIMAL 'SYS.TEMP.24' $
          DECLARE DECIMAL 'SYS.TEMP.25' $
          DECLARE DECIMAL 'SYS.TEMP.26' $
          DECLARE DECIMAL 'SYS.TEMP.27' $
          DECLARE DECIMAL 'SYS.TEMP.28' $
          DECLARE DECIMAL 'SYS.TEMP.29' $
          DECLARE DECIMAL 'SYS.TEMP.30' $
          DECLARE DECIMAL 'SYS.TEMP.31' $
          DECLARE DECIMAL 'SYS.TEMP.32' $
C $
C     USER DEFINED ATE FUNCTIONS $
C $
          DEFINE PROCEDURE 'OHMMETER' $
          DECLARE DECIMAL, 'OHMMETER.PRM01','OHMMETER.CNX01','OHMMETER.CNX02',
```

FIGURE A.7: (continued)

```
                        'OHMMETER.RES'$
          MEASURE (RES 'OHMMETER.PRM01' OHM), IMPEDANCE,
            REF-VOLTAGE 1 V,
            CNX HI 'OHMMETER.CNX01'
                LO 'OHMMETER.CNX02' $
        END 'OHMMETER' $
C $
        DEFINE PROCEDURE 'PSUPPLY' $
        DECLARE DECIMAL, 'PSUPPLY.PRM01','PSUPPLY.CNX01','PSUPPLY.CNX02',
                        'PSUPPLY.RES'$
          APPLY DC-SIGNAL DC2A,
            VOLTAGE 'PSUPPLY.PRM01' V,
            CNX HI 'PSUPPLY.CNX01'
                LO 'PSUPPLY.CNX02' $
        END 'PSUPPLY' $
C $
        DEFINE PROCEDURE 'VOLTMETER' $
        DECLARE DECIMAL,'VOLTMETER.PRM01','VOLTMETER.CNX01','VOLTMETER.CNX02',
                        'VOLTMETER.RES'$
          MEASURE (VOLTAGE 'VOLTMETER.PRM01' V), DC-SIGNAL,
            CNX HI 'VOLTMETER.CNX01'
                LO 'VOLTMETER.CNX02' $
        END 'VOLTMETER' $
C     DIAGNOSES PROCS $
 1000   DEFINE PROCEDURE, 'INP_SHORTED' $
        RECORD "*** AFTER STEP  1000 ATTEMPTING DIAGNOSIS 'INP_SHORTED' "$
        'SYS.#TESTS IN CONJ'(1) = 'SYS.#TESTS IN CONJ'(1) - 1  $
        COMPARE 'SYS.#TESTS IN CONJ'(1), LE 0  $
        GOTO STEP  1010     IF NOGO $
        RECORD "*** AFTER STEP  1000 SELECT DIAGNOSIS 'INP_SHORTED' "$
        RECORD " !L R/T DC INPUT SHORTED J24-B/J24-C  !L AN/GRC-106 ",
        "DEFECTIVE.  CHECK PRINTOUTS FOR DEFECTS. !L PRESS STOP." $
        'SYS.DIAG-FLAG'(1) = 'SYS.SELECTED' $
 1010   END 'INP_SHORTED' $
C $
 1100   DEFINE PROCEDURE, 'DISPLAY' $
        RECORD "*** AFTER STEP  1100 ATTEMPTING DIAGNOSIS 'DISPLAY' "$
        'SYS.#TESTS IN CONJ'(2) = 'SYS.#TESTS IN CONJ'(2) - 1  $
        COMPARE 'SYS.#TESTS IN CONJ'(2), LE 0  $
        GOTO STEP  1110     IF NOGO $
        RECORD "*** AFTER STEP  1100 SELECT DIAGNOSIS 'DISPLAY' "$
        RECORD " !L (P1):   (P2) " $
        'SYS.DIAG-FLAG'(2) = 'SYS.SELECTED' $
 1110   END 'DISPLAY' $
C $
 1200   DEFINE PROCEDURE, 'SHOW_MEAS' $
        RECORD "*** AFTER STEP  1200 ATTEMPTING DIAGNOSIS 'SHOW_MEAS' "$
        'SYS.#TESTS IN CONJ'(3) = 'SYS.#TESTS IN CONJ'(3) - 1  $
        COMPARE 'SYS.#TESTS IN CONJ'(3), LE 0  $
        GOTO STEP  1210     IF NOGO $
        RECORD "*** AFTER STEP  1200 SELECT DIAGNOSIS 'SHOW_MEAS' "$
        RECORD " !L (P1):   (P2) " $
        'SYS.DIAG-FLAG'(3) = 'SYS.SELECTED' $
 1210   END 'SHOW_MEAS' $
```

FIGURE A.7: (continued)

```
C $
 1300    DEFINE PROCEDURE, 'FREQ_TOL' $
         RECORD "*** AFTER STEP  1300 ATTEMPTING DIAGNOSIS 'FREQ_TOL' "$
         'SYS.#TESTS IN CONJ'(4) = 'SYS.#TESTS IN CONJ'(4) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(4), LE 0  $
         GOTO STEP  1310     IF NOGO $
         RECORD "*** AFTER STEP  1300 SELECT DIAGNOSIS 'FREQ_TOL' "$
         RECORD " !L (C) DEFECTIVE. !L 5.0 MHZ STD. OUT OF (P) TOLERA",
         "NCE." $
         'SYS.DIAG-FLAG'(4) = 'SYS.SELECTED' $
 1310    END 'FREQ_TOL' $
C $
 1400    DEFINE PROCEDURE, 'DISTORT_PRNT' $
         RECORD "*** AFTER STEP  1400 ATTEMPTING DIAGNOSIS 'DISTORT_PRNT' "$
         'SYS.#TESTS IN CONJ'(5) = 'SYS.#TESTS IN CONJ'(5) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(5), LE 0  $
         GOTO STEP  1410     IF NOGO $
         RECORD "*** AFTER STEP  1400 SELECT DIAGNOSIS 'DISTORT_PRNT' "$
         RECORD " !L (P1):   (P2) " $
         'SYS.DIAG-FLAG'(5) = 'SYS.SELECTED' $
 1410    END 'DISTORT_PRNT' $
C $
 1500    DEFINE PROCEDURE, 'HI_DISTORTIO' $
         RECORD "*** AFTER STEP  1500 ATTEMPTING DIAGNOSIS 'HI_DISTORTIO' "$
         'SYS.#TESTS IN CONJ'(6) = 'SYS.#TESTS IN CONJ'(6) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(6), LE 0  $
         GOTO STEP  1510     IF NOGO $
         RECORD "*** AFTER STEP  1500 SELECT DIAGNOSIS 'HI_DISTORTIO' "$
         RECORD " !L (P1) AUDIO DISTORTION GREATER THAN (P2) PERCENT.",
         "" $
         'SYS.DIAG-FLAG'(6) = 'SYS.SELECTED' $
 1510    END 'HI_DISTORTIO' $
C $
 1600    DEFINE PROCEDURE, 'WAIT' $
         RECORD "*** AFTER STEP  1600 ATTEMPTING DIAGNOSIS 'WAIT' "$
         'SYS.#TESTS IN CONJ'(7) = 'SYS.#TESTS IN CONJ'(7) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(7), LE 0  $
         GOTO STEP  1610     IF NOGO $
         RECORD "*** AFTER STEP  1600 SELECT DIAGNOSIS 'WAIT' "$
         RECORD " !L TUNE RECEIVER:  MC & KC CONTROLS TO 250000. !L A",
         "DJUST AUDIC GAIN CONTROL FOR 2.2 TO 2.8 VAC !L (2.5 VAC NOMI",
         "NAL).    PRESS  YES." $
         WAIT-FOR MANUAL-INTERVENTION $
         'SYS.DIAG-FLAG'(7) = 'SYS.SELECTED' $
 1610    END 'WAIT' $
C $
 1700    DEFINE PROCEDURE, 'VRMS_PRNT' $
         RECORD "*** AFTER STEP  1700 ATTEMPTING DIAGNOSIS 'VRMS_PRNT' "$
         'SYS.#TESTS IN CONJ'(8) = 'SYS.#TESTS IN CONJ'(8) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(8), LE 0  $
         GOTO STEP  1710     IF NOGO $
         RECORD "*** AFTER STEP  1700 SELECT DIAGNOSIS 'VRMS_PRNT' "$
         RECORD " !L (P1):   (P2) " $
         'SYS.DIAG-FLAG'(8) = 'SYS.SELECTED' $
```

FIGURE A.7: (continued)

```
 1710    END 'VRMS_PRNT' $
C $
 1800    DEFINE PROCEDURE, 'VRMS_FAILED' $
         RECORD "*** AFTER STEP  1800 ATTEMPTING DIAGNOSIS 'VRMS_FAILED' "$
         'SYS.#TESTS IN CONJ'(9) = 'SYS.#TESTS IN CONJ'(9) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(9), LE 0  $
         GOTO STEP  1810      IF NOGO $
         RECORD "*** AFTER STEP  1800 SELECT DIAGNOSIS 'VRMS_FAILED' "$
         RECORD " !L 10 MW DISTORTION REFERENCE VOLTAGE FAILED." $
         'SYS.DIAG-FLAG'(9) = 'SYS.SELECTED' $
 1810    END 'VRMS_FAILED' $
C $
 1900    DEFINE PROCEDURE, 'GET_DELAY_TI' $
         RECORD "*** AFTER STEP  1900 ATTEMPTING DIAGNOSIS 'GET_DELAY_TI' "$
         'SYS.#TESTS IN CONJ'(10) = 'SYS.#TESTS IN CONJ'(10) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(10), LE 0  $
         GOTO STEP  1910      IF NOGO $
         RECORD "*** AFTER STEP  1900 SELECT DIAGNOSIS 'GET_DELAY_TI' "$
         RECORD " !L IF A 12 MINUTE WARMUP IS DESIRED, ENTER IN 720; ",
         "!L  OTHERWISE, KEY IN 60.   PRESS YES." $
         INPUT 'DELAY-TIME.' $
         'SYS.DIAG-FLAG'(10) = 'SYS.SELECTED' $
 1910    END 'GET_DELAY_TI' $
C $
 2000    DEFINE PROCEDURE, 'FREQ_TOL_FAI' $
         RECORD "*** AFTER STEP  2000 ATTEMPTING DIAGNOSIS 'FREQ_TOL_FAI' "$
         'SYS.#TESTS IN CONJ'(11) = 'SYS.#TESTS IN CONJ'(11) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(11), LE 0  $
         GOTO STEP  2010      IF NOGO $
         RECORD "*** AFTER STEP  2000 SELECT DIAGNOSIS 'FREQ_TOL_FAI' "$
         RECORD " !L (C) DEFECTIVE. !L 5.0 MHZ STD. OUT OF (P) TOLERA",
         "NCE." $
         'SYS.DIAG-FLAG'(11) = 'SYS.SELECTED' $
 2010    END 'FREQ_TOL_FAI' $
C $
 2100    DEFINE PROCEDURE, 'FREQ_PRNT' $
         RECORD "*** AFTER STEP  2100 ATTEMPTING DIAGNOSIS 'FREQ_PRNT' "$
         'SYS.#TESTS IN CONJ'(12) = 'SYS.#TESTS IN CONJ'(12) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(12), LE 0  $
         GOTO STEP  2110      IF NOGO $
         RECORD "*** AFTER STEP  2100 SELECT DIAGNOSIS 'FREQ_PRNT' "$
         RECORD " !L (P1):   (P2) " $
         'SYS.DIAG-FLAG'(12) = 'SYS.SELECTED' $
 2110    END 'FREQ_PRNT' $
C $
 2200    DEFINE PROCEDURE, 'AUDIO_DISTOR' $
         RECORD "*** AFTER STEP  2200 ATTEMPTING DIAGNOSIS 'AUDIO_DISTOR' "$
         'SYS.#TESTS IN CONJ'(13) = 'SYS.#TESTS IN CONJ'(13) - 1  $
         COMPARE 'SYS.#TESTS IN CONJ'(13), LE 0  $
         GOTO STEP  2210      IF NOGO $
         RECORD "*** AFTER STEP  2200 SELECT DIAGNOSIS 'AUDIO_DISTOR' "$
         RECORD " !L (P1) AUDIO DISTORTION GREATER THAN (P2) PERCENT.",
         "" $
         'SYS.DIAG-FLAG'(13) = 'SYS.SELECTED' $
```

FIGURE A.7: (continued)

```
 2210    END 'AUDIO_DISTOR' $
C $
C    TEST PROCS $
C $
 2300    DEFINE PROCEDURE, 'DC-INPUT'$
         RECORD "*** AFTER STEP  2300 !L *** ENTERED TEST PROCEDURE ",
                "'DC-INPUT' AT ",'PRT.TIME'$
         DECLARE DECIMAL,  'MRES.1' $
         'SYS.FLAG' = 'SYS.TRUE'$
         PERFORM 'GET.TIME ' $
         'SYS.S-TIME' = 'SYS.TIME' $
         RECORD "*** AFTER STEP  2300    ",
         "DIAGNOSIS SELECTED UNCONDITIONALLY $
         PERFORM 'DISPLAY'  $
C    ATLAS CODE FOR WAVEFORM SYS.M-W0001 FOLLOWS: $
         RECORD "*** AFTER STEP  2300     BEGIN CONJUNCTION SYS.M-W0001 "$
C  THE  CONNECTION PTS  $
         'OHMMETER.CNX01' = 'J24-B' $
         'OHMMETER.CNX02' = 'J24-C' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C   FUNCTION CALL  $
         PERFORM 'MRES' $
C   COPY BACK THE PARAMETERS AND RESULT  $
         'SYS.TEMP.01' = 'MRES.RES' $
C   END OF FN CALL  $
         RECORD "*** AFTER STEP  2300     END CONJUNCTION SYS.M-W0001 "$
C  DIMENSION OF VDE IS OHM $
         'OHMMETER.PRM01' = 'SYS.TEMP.01' $
         PERFORM 'OHMMETER' $
C   COPY BACK THE PARAMETERS AND RESULT  $
         'SYS.TEMP.02' = 'OHMMETER.RES' $
C   END OF FN CALL  $
         RECORD "*** AFTER STEP  2300     END CONJUNCTION SYS.M-W0001 "$
C    $
C    ATLAS CODE FOR WAVEFORM SYS.M-W0002 FOLLOWS: $
         RECORD "*** AFTER STEP  2300     BEGIN ASSERTION SYS.M-W0002 "$
C   FUNCTION CALL  $
         PERFORM 'MRES' $
C   COPY BACK THE PARAMETERS AND RESULT  $
         'SYS.TEMP.03' = 'MRES.RES' $
C   END OF FN CALL  $
         RECORD "*** AFTER STEP  2300     END ASSERTION SYS.M-W0002 "$
         COMPARE 'SYS.TEMP.03',  GT  1.00000E+02 $
         GOTO STEP  2310     IF GO $
         'SYS.FLAG' = 'SYS.FALSE' $
         RECORD "*** AFTER STEP  2300     ASSERTION ",
                "'SYS.M-W0002' EVALUATED       TO FALSE $
C    $
 2310    COMPARE 'SYS.FLAG'   ,EQ 'SYS.TRUE'$
         GOTO STEP  2320     IF NOGO $
         RECORD "*** AFTER STEP  2310 DIAGNOSIS SELECTED BY TRUE OUTCOME "$
         GOTO STEP  2330    $
 2320    RECORD "*** AFTER STEP  2310 DIAGNOSIS SELECTED BY FALSE OUTCOME "$
```

FIGURE A.7: (continued)

```
          PERFORM 'INF_SHORTED' $
 2330     'SYS.TEST-FLAG'(1) = 'SYS.TESTED'$
          RECORD "*** AFTER STEP  2330 RETURNING FROM TEST PROCEDURE ",
          "'DC-INPUT' AT ",'PRT.TIME'$
 2340     END 'DC-INPUT'$
C     $
 2400     DEFINE PROCEDURE, 'AMPL'$
          RECORD "*** AFTER STEP  2400 !L *** ENTERED TEST PROCEDURE ",
                 "'AMPL' AT ",'PRT.TIME'$
          DECLARE DECIMAL,  'V-SIN.2' $
          'SYS.FLAG' = 'SYS.TRUE'$
          PERFORM 'GET.TIME ' $
          'SYS.S-TIME ' = 'SYS.TIME' $
          RECORD "*** AFTER STEP  2400    ",
          "DIAGNOSIS SELECTED UNCONDITIONALLY $
          PERFORM 'SHOW_MEAS'  $
C     ATLAS CODE FOR WAVEFORM SYS.M-W0001 FOLLOWS: $
          RECORD "*** AFTER STEP  2400     BEGIN ASSERTION SYS.M-W0001 "$
C    FUNCTION CALL  $
          PERFORM 'V-SIN' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.04' = 'V-SIN.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2400     END ASSERTION SYS.M-W0001 "$
          'SYS.TEMP.05' =  5.99999E-02 $
          COMPARE 'SYS.TEMP.04', UL  2.59999E-01 +
          'SYS.TEMP.05' LL  2.59999E-01 - 'SYS.TEMP.05' $
          GOTO STEP  2410     IF GO $
          'SYS.FLAG' = 'SYS.FALSE' $
          RECORD "*** AFTER STEP  2400     ASSERTION ",
                 "'SYS.M-W0001' EVALUATED       TO FALSE $
C     $
 2410     COMPARE 'SYS.FLAG'   ,EQ 'SYS.TRUE'$
          GOTO STEP  2420     IF NOGO $
          RECORD "*** AFTER STEP  2410 DIAGNOSIS SELECTED BY TRUE OUTCOME "$
          GOTO STEP  2430    $
 2420     RECORD "*** AFTER STEP  2410 DIAGNOSIS SELECTED BY FALSE OUTCOME "$
          PERFORM 'FREQ_TOL' $
 2430     'SYS.TEST-FLAG'(2) = 'SYS.TESTED'$
          RECORD "*** AFTER STEP  2430 RETURNING FROM TEST PROCEDURE ",
          "'AMPL' AT ",'PRT.TIME'$
 2440     END 'AMPL'$
C     $
 2500     DEFINE PROCEDURE, 'DISTORT-2W'$
          RECORD "*** AFTER STEP  2500 !L *** ENTERED TEST PROCEDURE ",
                 "'DISTORT-2W' AT ",'PRT.TIME'$
          DECLARE DECIMAL,  'M-DISTORT.3' $
          'SYS.FLAG' = 'SYS.TRUE'$
          PERFORM 'GET.TIME ' $
          'SYS.S-TIME ' = 'SYS.TIME' $
          RECORD "*** AFTER STEP  2500    ",
          "DIAGNOSIS SELECTED UNCONDITIONALLY $
          PERFORM 'DISTORT_PRNT'  $
C     ATLAS CODE FOR WAVEFORM SYS.S-W0001 FOLLOWS: $
```

FIGURE A.7: (continued)

```
        RECORD "*** AFTER STEP  2500     BEGIN CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $
        'PWR-SUPPLY.CNX01' = 'J24-B' $
        'PWR-SUPPLY.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS VOLT $
        'PWR-SUPPLY.PRM01' =  2.75000E+01 $
        PERFORM 'PWR-SUPPLY' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.06' = 'PWR-SUPPLY.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2500     END CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $ .
        'SIGNAL-AM.CNX01' = 'J16' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS MHZ $
        'SIGNAL-AM.PRM01' =  2.50019E+01 $
C  DIMENSION OF VDE IS DB $
        'SIGNAL-AM.PRM02' = ( 0.00000E+00 +   1.30000E+01) $
C  DIMENSION OF VDE IS % $
        'SIGNAL-AM.PRM03' =  0.00000E+00 $
C  DIMENSION OF VDE IS KHZ $
        'SIGNAL-AM.PRM04' =  1.00000E+00 $
        PERFORM 'SIGNAL-AM' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.07' = 'SIGNAL-AM.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2500     END CONJUNCTION SYS.S-W0001 "$
C    $
C    ATLAS CODE FOR WAVEFORM SYS.M-W0001 FOLLOWS: $
        RECORD "*** AFTER STEP  2500     BEGIN CONJUNCTION SYS.M-W0001 "$
C  THE  CONNECTION PTS  $
        'DISTORTION.CNX01' = 'J19-L' $
        'DISTORTION.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C   FUNCTION CALL  $
        PERFORM 'M-DISTORT' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.08' = 'M-DISTORT.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2500     END CONJUNCTION SYS.M-W0001 "$
C  DIMENSION OF VDE IS % $
        'DISTORTION.PRM01' = 'SYS.TEMP.08' $
C  DIMENSION OF VDE IS KHZ $
        'DISTORTION.PRM02' =  1.00000E+00 $
        PERFORM 'DISTORTION' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.09' = 'DISTORTION.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2500     END CONJUNCTION SYS.M-W0001 "$
C    $
```

FIGURE A.7: (continued)

```
C     ATLAS CODE FOR WAVEFORM SYS.M-W0002 FOLLOWS: $
      RECORD "*** AFTER STEP  2500     BEGIN ASSERTION SYS.M-W0002 "$
C    FUNCTION CALL  $
      PERFORM 'M-DISTORT' $
C    COPY BACK THE PARAMETERS AND RESULT  $
      'SYS.TEMP.10' = 'M-DISTORT.RES' $
C    END OF FN CALL  $
      RECORD "*** AFTER STEP  2500     END ASSERTION SYS.M-W0002 "$
      COMPARE 'SYS.TEMP.10',  LE  5.00000E+00 $
      GOTO STEP  2510     IF GO $
      'SYS.FLAG' = 'SYS.FALSE' $
      RECORD "*** AFTER STEP  2500     ASSERTION ",
             "'SYS.M-W0002' EVALUATED       TO FALSE $
C    $
 2510 COMPARE 'SYS.FLAG'   ,EQ 'SYS.TRUE'$
      GOTO STEP  2520     IF NOGO $
      RECORD "*** AFTER STEP  2510 DIAGNOSIS SELECTED BY TRUE OUTCOME "$
      GOTO STEP  2530    $
 2520 RECORD "*** AFTER STEP  2510 DIAGNOSIS SELECTED BY FALSE OUTCOME "$
      PERFORM 'HI_DISTORTIO' $
 2530 'SYS.TEST-FLAG'(3) = 'SYS.TESTED'$
      RECORD "*** AFTER STEP  2530 RETURNING FROM TEST PROCEDURE ",
      "'DISTORT-2W' AT ",'PRT.TIME'$
 2540 END 'DISTORT-2W'$
C    $
 2600 DEFINE PROCEDURE, 'DISTORT-VOLT'$
      RECORD "*** AFTER STEP  2600 !L *** ENTERED TEST PROCEDURE ",
             "'DISTORT-VOLT' AT ",'PRT.TIME'$
      DECLARE DECIMAL,  'VRMS.4' $
      'SYS.FLAG' = 'SYS.TRUE'$
      PERFORM 'GET.TIME ' $
      'SYS.S-TIME' = 'SYS.TIME' $
      RECORD "*** AFTER STEP  2600    ",
      "DIAGNOSIS SELECTED UNCONDITIONALLY $
      PERFORM 'WAIT'  $
      RECORD "*** AFTER STEP  2600    ",
      "DIAGNOSIS SELECTED UNCONDITIONALLY $
      PERFORM 'VRMS_PRNT'  $
C     ATLAS CODE FOR WAVEFORM SYS.S-W0001 FOLLOWS: $
      RECORD "*** AFTER STEP  2600     BEGIN CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $
      'PWR-SUPPLY.CNX01' = 'J24-B' $
      'PWR-SUPPLY.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS VOLT $
      'PWR-SUPPLY.PRM01' =  2.75000E+01 $
      PERFORM 'PWR-SUPPLY' $
C    COPY BACK THE PARAMETERS AND RESULT  $
      'SYS.TEMP.11' = 'PWR-SUPPLY.RES' $
C    END OF FN CALL  $
      RECORD "*** AFTER STEP  2600     END CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $
      'SIGNAL-AM.CNX01' = 'J16' $
```

FIGURE A.7: (continued)

```
        COMPARE 'SYS.TEMP.16', GE  2.19999E+00 $
        GOTO STEP  2610     IF GO $
        'SYS.FLAG' = 'SYS.FALSE' $
        RECORD "*** AFTER STEP  2600     ASSERTION ",
               "'A1' EVALUATED       TO FALSE $
C    $
C    ATLAS CODE FOR WAVEFORM A2 FOLLOWS: $
 2610   RECORD "*** AFTER STEP  2600     BEGIN ASSERTION A2 "$
C   FUNCTION CALL  $
        PERFORM 'VRMS' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.17' = 'VRMS.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2610     END ASSERTION A2 "$
        COMPARE 'SYS.TEMP.17', LE  2.79999E+00 $
        GOTO STEP  2620     IF GO $
        'SYS.FLAG' = 'SYS.FALSE' $
        RECORD "*** AFTER STEP  2610     ASSERTION ",
               "'A2' EVALUATED       TO FALSE $
C    $
 2620   COMPARE 'SYS.FLAG'   ,EQ 'SYS.TRUE'$
        GOTO STEP  2630     IF NOGO $
        RECORD "*** AFTER STEP  2620 DIAGNOSIS SELECTED BY TRUE OUTCOME "$
        GOTO STEP  2640    $
 2630   RECORD "*** AFTER STEP  2620 DIAGNOSIS SELECTED BY FALSE OUTCOME "$
        PERFORM 'VRMS_FAILED' $
 2640   'SYS.TEST-FLAG'(4) = 'SYS.TESTED'$
        RECORD "*** AFTER STEP  2640 RETURNING FROM TEST PROCEDURE ",
        "'DISTORT-VOLT' AT ",'PRT.TIME'$
 2650   END 'DISTORT-VOLT'$
C    $
 2700   DEFINE PROCEDURE, 'FREQ'$
        RECORD "*** AFTER STEP  2700 !L *** ENTERED TEST PROCEDURE ",
               "'FREQ' AT ",'PRT.TIME'$
        DECLARE DECIMAL,  'DELAY-TIME.5' $
        DECLARE DECIMAL,  'FREQ.5' $
        DECLARE DECIMAL,  'V-SIN.5' $
        'SYS.FLAG' = 'SYS.TRUE'$
        PERFORM 'GET.TIME ' $
        'SYS.S-TIME' = 'SYS.TIME' $
        RECORD "*** AFTER STEP  2700     ",
        "DIAGNOSIS SELECTED UNCONDITIONALLY $
        PERFORM 'GET_DELAY_TI'  $
        RECORD "*** AFTER STEP  2700     ",
        "DIAGNOSIS SELECTED UNCONDITIONALLY $
        PERFORM 'FREQ_PRNT'  $
C    ATLAS CODE FOR WAVEFORM SYS.S-W0001 FOLLOWS: $
        RECORD "*** AFTER STEP  2700     BEGIN CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $
        'PWR-SUPPLY.CNX01' = 'J24-8' $
        'PWR-SUPPLY.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS VOLT $
```

FIGURE A.7: (continued)

```
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS MHZ $
        'SIGNAL-AM.PRM01' =  2.50019E+01 $
C  DIMENSION OF VDE IS DB $
        'SIGNAL-AM.PRM02' = ( 0.00000E+00 +   1.30000E+01) $
C  DIMENSION OF VDE IS % $
        'SIGNAL-AM.PRM03' =  0.00000E+00 $
C  DIMENSION OF VDE IS KHZ $
        'SIGNAL-AM.PRM04' =  1.00000E+00 $
        PERFORM 'SIGNAL-AM' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.12' = 'SIGNAL-AM.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2600     END CONJUNCTION SYS.S-W0001 "$
C    $
C    ATLAS CODE FOR WAVEFORM SYS.M-W0001 FOLLOWS: $
        RECORD "*** AFTER STEP  2600     BEGIN CONJUNCTION SYS.M-W0001 "$
C  THE  CONNECTION PTS  $
        'SINE-WAVE.CNX01' = 'J19-A' $
        'SINE-WAVE.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C   FUNCTION CALL  $
        PERFORM 'VRMS' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.13' = 'VRMS.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2600     END CONJUNCTION SYS.M-W0001 "$
C  DIMENSION OF VDE IS VOLT $
        'SINE-WAVE.PRM01' = 'SYS.TEMP.13' $
C   FUNCTION CALL  $
        PERFORM 'SYS.M-W0001' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.14' = 'SYS.M-W0001.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2600     END CONJUNCTION SYS.M-W0001 "$
        'SINE-WAVE.PRM02' = 'SYS.TEMP.14' $
C  DIMENSION OF VDE IS SEC $
        'SINE-WAVE.PRM03' =  0.00000E+00 $
        PERFORM 'SINE-WAVE' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.15' = 'SINE-WAVE.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2600     END CONJUNCTION SYS.M-W0001 "$
C    $
C    ATLAS CODE FOR WAVEFORM A1 FOLLOWS: $
        RECORD "*** AFTER STEP  2600     BEGIN ASSERTION A1 "$
C   FUNCTION CALL  $
        PERFORM 'VRMS' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.16' = 'VRMS.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2600     END ASSERTION A1 "$
```

FIGURE A.7: (continued)

```
          'PWR-SUPPLY.PRM01' =  2.75000E+01 $
          PERFORM 'PWR-SUPPLY' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.18' = 'PWR-SUPPLY.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2700      END CONJUNCTION SYS.S-W0001 "$
C     $
C     ATLAS CODE FOR WAVEFORM SYS.M-W0001 FOLLOWS: $
          RECORD "*** AFTER STEP  2700      BEGIN CONJUNCTION SYS.M-W0001 "$
C   THE  CONNECTION PTS  $
          'SINE-WAVE.CNX01' = 'J22' $
          'SINE-WAVE.CNX02' = 'GND' $
C   THE STIMULI/MEAS FUNCTION  $
C    FUNCTION CALL  $
C    FUNCTION CALL  $
          PERFORM 'V-SIN' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.19' = 'V-SIN.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2700      END CONJUNCTION SYS.M-W0001 "$
C   DIMENSION OF VDE IS VOLT $
          'SINE-WAVE.PRM01' = 'SYS.TEMP.19' $
C    FUNCTION CALL  $
          PERFORM 'FREQ' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.20' = 'FREQ.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2700      END CONJUNCTION SYS.M-W0001 "$
C   DIMENSION OF VDE IS HZ $
          'SINE-WAVE.PRM02' = 'SYS.TEMP.20' $
C    FUNCTION CALL  $
          PERFORM 'DELAY-TIME' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.21' = 'DELAY-TIME.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2700      END CONJUNCTION SYS.M-W0001 "$
C   DIMENSION OF VDE IS SEC $
          'SINE-WAVE.PRM03' = 'SYS.TEMP.21' $
          PERFORM 'SINE-WAVE' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.22' = 'SINE-WAVE.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2700      END CONJUNCTION SYS.M-W0001 "$
C     $
C     ATLAS CODE FOR WAVEFORM SYS.M-W0002 FOLLOWS: $
C    FUNCTION CALL  $
          PERFORM 'DELAY-TIME' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.23' = 'DELAY-TIME.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2700      END CONJUNCTION SYS.M-W0002 "$
          COMPARE ('SYS.TEMP.23' EQ  6.00000E+01)WITH 0 $
C    FUNCTION CALL  $
C    COPY BACK THE PARAMETERS AND RESULT  $
```

FIGURE A.7: (continued)

```
C    END OF FN CALL  $
C    FUNCTION CALL  $
C    COPY BACK THE PARAMETERS AND RESULT  $
C    END OF FN CALL  $
          GOTO  2720      IF NOGO $
          RECORD "*** AFTER STEP  2700      BEGIN ASSERTION SYS.M-W0002 "$
          PERFORM 'FREQ' $
          'SYS.TEMP.24' = 'FREQ.RES' $
          RECORD "*** AFTER STEP  2700      END ASSERTION SYS.M-W0002 "$
          'SYS.TEMP.25' =  6.00000E+01 $
          COMPARE 'SYS.TEMP.24', UL  5.00000E+06 +
          'SYS.TEMP.25' LL  5.00000E+06 - 'SYS.TEMP.25' $
          GOTO STEP  2710      IF GO $
          'SYS.FLAG' = 'SYS.FALSE' $
          RECORD "*** AFTER STEP  2700      ASSERTION ",
                 "'SYS.M-W0002' EVALUATED        TO FALSE $
 2710     GOTO  2730      $
 2720     RECORD "*** AFTER STEP  2710      BEGIN ASSERTION SYS.M-W0002 "$
          PERFORM 'FREQ' $
          'SYS.TEMP.26' = 'FREQ.RES' $
          RECORD "*** AFTER STEP  2720      END ASSERTION SYS.M-W0002 "$
          'SYS.TEMP.27' =  2.50000E+00 $
          COMPARE 'SYS.TEMP.26', UL  5.00000E+06 +
          'SYS.TEMP.27' LL  5.00000E+06 - 'SYS.TEMP.27' $
          GOTO STEP  2730      IF GO $
          'SYS.FLAG' = 'SYS.FALSE' $
          RECORD "*** AFTER STEP  2720      ASSERTION ",
                 "'SYS.M-W0002' EVALUATED        TO FALSE $
C    $
 2730     COMPARE 'SYS.FLAG'    ,EQ 'SYS.TRUE'$
          GOTO STEP  2740      IF NOGO $
          RECORD "*** AFTER STEP  2730 DIAGNOSIS SELECTED BY TRUE OUTCOME "$
          GOTO STEP  2750    $
 2740     RECORD "*** AFTER STEP  2730 DIAGNOSIS SELECTED BY FALSE OUTCOME "$
          PERFORM 'FREQ_TOL_FAI' $
 2750     'SYS.TEST-FLAG'(5) = 'SYS.TESTED'$
          RECORD "*** AFTER STEP  2750 RETURNING FROM TEST PROCEDURE ",
          "'FREQ' AT ",'PRT.TIME'$
 2760     END 'FREQ'$
C    $
 2800     DEFINE PROCEDURE, 'DISTORT-10MW'$
          RECORD "*** AFTER STEP  2800 !L *** ENTERED TEST PROCEDURE ",
                 "'DISTORT-10MW' AT ",'PRT.TIME'$
          DECLARE DECIMAL,  'M-DISTORT.6' $
          'SYS.FLAG' = 'SYS.TRUE'$
          PERFORM 'GET.TIME ' $
          'SYS.S-TIME' = 'SYS.TIME' $
          RECORD "*** AFTER STEP  2800    ",
          "DIAGNOSIS SELECTED UNCONDITIONALLY $
          PERFORM 'DISTORT_PRNT'  $
C     ATLAS CODE FOR WAVEFORM SYS.S-W0001 FOLLOWS: $
          RECORD "*** AFTER STEP  2800      BEGIN CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $
          'PWR-SUPPLY.CNX01' = 'J24-8' $
```

FIGURE A.7: (continued)

```
        'PWR-SUPPLY.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS VOLT $
        'PWR-SUPPLY.PRM01' =  2.75000E+01 $
        PERFORM 'PWR-SUPPLY' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.28' = 'PWR-SUPPLY.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2800     END CONJUNCTION SYS.S-W0001 "$
C  THE  CONNECTION PTS  $
        'SIGNAL-AM.CNX01' = 'J16' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C  DIMENSION OF VDE IS MHZ $
        'SIGNAL-AM.PRM01' =  2.50019E+01 $
C  DIMENSION OF VDE IS DB $
        'SIGNAL-AM.PRM02' = ( 0.00000E+00 +   1.30000E+01) $
C  DIMENSION OF VDE IS % $
        'SIGNAL-AM.PRM03' =  0.00000E+00 $
C  DIMENSION OF VDE IS KHZ $
        'SIGNAL-AM.PRM04' =  1.00000E+00 $
        PERFORM 'SIGNAL-AM' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.29' = 'SIGNAL-AM.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2800     END CONJUNCTION SYS.S-W0001 "$
C    $
C    ATLAS CODE FOR WAVEFORM SYS.M-W0001 FOLLOWS: $
        RECORD "*** AFTER STEP  2800     BEGIN CONJUNCTION SYS.M-W0001 "$
C  THE  CONNECTION PTS  $
        'DISTORTION.CNX01' = 'J19-A' $
        'DISTORTION.CNX02' = 'GND' $
C  THE STIMULI/MEAS FUNCTION  $
C   FUNCTION CALL  $
C   FUNCTION CALL  $
        PERFORM 'M-DISTORT' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.30' = 'M-DISTORT.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2800     END CONJUNCTION SYS.M-W0001 "$
C  DIMENSION OF VDE IS % $
        'DISTORTION.PRM01' = 'SYS.TEMP.30' $
C  DIMENSION OF VDE IS KHZ $
        'DISTORTION.PRM02' =  2.00000E+00 $
        PERFORM 'DISTORTION' $
C   COPY BACK THE PARAMETERS AND RESULT  $
        'SYS.TEMP.31' = 'DISTORTION.RES' $
C   END OF FN CALL  $
        RECORD "*** AFTER STEP  2800     END CONJUNCTION SYS.M-W0001 "$
C    $
C    ATLAS CODE FOR WAVEFORM SYS.M-W0002 FOLLOWS: $
        RECORD "*** AFTER STEP  2800     BEGIN ASSERTION SYS.M-W0002 "$
C   FUNCTION CALL  $
```

FIGURE A.7: (continued)

```
          PERFORM 'M-DISTORT' $
C    COPY BACK THE PARAMETERS AND RESULT  $
          'SYS.TEMP.32' = 'M-DISTORT.RES' $
C    END OF FN CALL  $
          RECORD "*** AFTER STEP  2800     END ASSERTION SYS.M-W0002 "$
          COMPARE 'SYS.TEMP.32',  LE  3.00000E+00 $
          GOTO STEP  2810     IF GO $
          'SYS.FLAG' = 'SYS.FALSE' $
          RECORD "*** AFTER STEP  2800     ASSERTION ",
                 "'SYS.M-W0002' EVALUATED       TO FALSE $
C     $
 2810     COMPARE 'SYS.FLAG'   ,EQ 'SYS.TRUE'$
          GOTO STEP  2820     IF NOGO $
          RECORD "*** AFTER STEP  2810 DIAGNOSIS SELECTED BY TRUE OUTCOME "$
          GOTO STEP  2830    $
 2820     RECORD "*** AFTER STEP  2810 DIAGNOSIS SELECTED BY FALSE OUTCOME "$
          PERFORM 'AUDIO_DISTOR' $
 2830     'SYS.TEST-FLAG'(6) = 'SYS.TESTED'$
          RECORD "*** AFTER STEP  2830 RETURNING FROM TEST PROCEDURE ",
          "'DISTORT-10MW' AT ",'PRT.TIME'$
 2840     END 'DISTORT-10MW'$
C     $
C     SYSTEM VARIABLE INITIALIZATION AND FIRST ENTRY POINT   $
E  2900     PERFORM 'GET.TIME'  $
          DISPLAY "!P TESTING STARTED FOR UUT MINIRADIOSET" $
          DISPLAY  'SYS.CLOCK'(4), "DATE ##/",'SYS.CLOCK'(5), "##/" ,
                   'SYS.CLOCK'(6), "##", 'PRT.TIME' $
          'SYS.TIM' = 'SYS.TIME' $
          FOR 'SYS.I' = 1 THRU 'SYS.#DIAGS' THEN $
             'SYS.DIAG-FLAG'('SYS.I')= 'SYS.NOT SELECTED' $
             'SYS.#TESTS IN CONJ'('SYS.I') = 0 $
          END FOR  $
          FOR 'SYS.I' = 1 THRU 'SYS.#TESTS' THEN $
              'SYS.TEST-FLAG'('SYS.I') = 'SYS.NOT TESTED' $
          END FOR $
C     BEGINNING OF TESTING       $
C     $
C     CONTROL PRECEDING THE CALL ON THE TEST MODULE 'DC_INPUT' $
 3000     'SYS.FLAG' = 'SYS.TRUE' $
 3010     COMPARE 'SYS.FLAG', EQ 'SYS.TRUE' $
          GOTO STEP  3100     IF NOGO $
          COMPARE 'SYS.TEST-FLAG'(1), EQ 'SYS.NOT TESTED' $
          GOTO STEP  3100     IF NOGO $
          PERFORM 'DC-INPUT' $
C $
C     CONTROL PRECEDING THE CALL ON THE TEST MODULE 'FREQ' $
 3100     'SYS.FLAG' = 'SYS.TRUE' $
          'SYS.FLAG' = 'SYS.FLAG' AND NOT('SYS.DIAG-FLAG'(1) XOR
                 'SYS.NOT SELECTED' ) $
          COMPARE 'SYS.TEST-FLAG'(5), NE 'SYS.SKIPPED' $
          GOTO STEP  3110     IF NOGO $
          COMPARE 'SYS.DIAG-FLAG'(1), EQ 'SYS.SELECTED' $
          GOTO STEP  3110     IF NOGO $
          'SYS.TEST-FLAG'(5) = 'SYS.SKIPPED' $
```

FIGURE A.7: (continued)

```
 3110    COMPARE 'SYS.FLAG', EQ 'SYS.TRUE' $
         GOTO STEP  3200     IF NOGO $
         COMPARE 'SYS.TEST-FLAG'(5), EQ 'SYS.NOT TESTED' $
         GOTO STEP  3200     IF NOGO $
         PERFORM 'FREQ' $
C $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'AMPL' $
 3200    'SYS.FLAG' = 'SYS.TRUE' $
         'SYS.FLAG' = 'SYS.FLAG' AND NOT('SYS.DIAG-FLAG'(1) XOR
                 'SYS.NOT SELECTED' ) $
         COMPARE 'SYS.TEST-FLAG'(2), NE 'SYS.SKIPPED' $
         GOTO STEP  3210     IF NOGO $
         COMPARE 'SYS.DIAG-FLAG'(1), EQ 'SYS.SELECTED' $
         GOTO STEP  3210     IF NOGO $
         'SYS.TEST-FLAG'(2) = 'SYS.SKIPPED' $
 3210    COMPARE 'SYS.FLAG', EQ 'SYS.TRUE' $
         GOTO STEP  3300     IF NOGO $
         COMPARE 'SYS.TEST-FLAG'(2), EQ 'SYS.NOT TESTED' $
         GOTO STEP  3300     IF NOGO $
         PERFORM 'AMPL' $
C $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'DISTORT_2W' $
 3300    'SYS.FLAG' = 'SYS.TRUE' $
 3310    COMPARE 'SYS.FLAG', EQ 'SYS.TRUE' $
         GOTO STEP  3400     IF NOGO $
         COMPARE 'SYS.TEST-FLAG'(3), EQ 'SYS.NOT TESTED' $
         GOTO STEP  3400     IF NOGO $
         PERFORM 'DISTORT-2W' $
C $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'DISTORT_VOLT' $
 3400    'SYS.FLAG' = 'SYS.TRUE' $
         'SYS.FLAG' = 'SYS.FLAG' AND NOT('SYS.DIAG-FLAG'(1) XOR
                 'SYS.NOT SELECTED' ) $
         COMPARE 'SYS.TEST-FLAG'(4), NE 'SYS.SKIPPED' $
         GOTO STEP  3410     IF NOGO $
         COMPARE 'SYS.DIAG-FLAG'(1), EQ 'SYS.SELECTED' $
         GOTO STEP  3410     IF NOGO $
         'SYS.TEST-FLAG'(4) = 'SYS.SKIPPED' $
 3410    COMPARE 'SYS.FLAG', EQ 'SYS.TRUE' $
         GOTO STEP  3500     IF NOGO $
         COMPARE 'SYS.TEST-FLAG'(4), EQ 'SYS.NOT TESTED' $
         GOTO STEP  3500     IF NOGO $
         PERFORM 'DISTORT-VOLT' $
C $
C    CONTROL PRECEDING THE CALL ON THE TEST MODULE 'DISTORT_10MW' $
 3500    'SYS.FLAG' = 'SYS.TRUE' $
         'SYS.FLAG' = 'SYS.FLAG' AND NOT('SYS.DIAG-FLAG'(1) XOR
                 'SYS.NOT SELECTED' ) $
         COMPARE 'SYS.TEST-FLAG'(6), NE 'SYS.SKIPPED' $
         GOTO STEP  3510     IF NOGO $
         COMPARE 'SYS.DIAG-FLAG'(1), EQ 'SYS.SELECTED' $
         GOTO STEP  3510     IF NOGO $
         'SYS.TEST-FLAG'(6) = 'SYS.SKIPPED' $
 3510    'SYS.FLAG' = 'SYS.FLAG' AND NOT('SYS.DIAG-FLAG'(9) XOR
```

FIGURE A.7: (continued)

```
            'SYS.NOT SELECTED' ) $
        COMPARE 'SYS.TEST-FLAG'(6), NE 'SYS.SKIPPED' $
        GOTO STEP  3520      IF NOGO $
        COMPARE 'SYS.DIAG-FLAG'(9), EQ 'SYS.SELECTED' $
        GOTO STEP  3520      IF NOGO $
        'SYS.TEST-FLAG'(6) = 'SYS.SKIPPED' $
 3520   COMPARE 'SYS.FLAG', EQ 'SYS.TRUE' $
        GOTO STEP  3600      IF NOGO $
        COMPARE 'SYS.TEST-FLAG'(6), EQ 'SYS.NOT TESTED' $
        GOTO STEP  3600      IF NOGO $
        PERFORM 'DISTORT-10MW' $
C $
 3600   PERFORM 'GET.TIME' $
        DISPLAY "NORMAL TERMINATION OF TESTING AT", 'PRT.TIME' $
        'SYS.TIM = 'SYS.TIME' - 'SYS.TIM' $
        'SYS.CLOCK'(1) = INT ('SYS.TIM'/3600) $
        'SYS.TIM' = 'SYS.TIM' - 3600*'SYS.CLOCK'(1) $
        'SYS.CLOCK'(2) = INT('SYS.TIM'/60) $
        'SYS.CLOCK'(3) = 'SYS.TIM' - 60*'SYS.CLOCK'(2) $
        DISPLAY "DURATION " , 'PRT.TIME' $
        FINISH $
        TERMINATE EQUATE PROGRAM 'MINIRADIOSET' $
```

FIGURE A.7: (continued)

APPENDIX B

NOPAL SYSTEM DATA STRUCTURES FOR THE ASSOCIATIVE MEMORY

This appendix illustrates the data structures used to encode all the eighteen types of statements (as listed in Table 4.9) 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 PL/I based structure. In the drawing, each cell represents a field in the data structure. The PL/I declaration for the structure is given below the drawings.

The storage entries link the structure containing actual data to the directory, and also contain list of variables occurring in the data. For example: Storage entry for a particular conjunction and assertion has its DATA field pointing to the structure WAVEFORM (which contains the actual data for the conjunction or assertion as the case may be). It also has indices pointing back into directory for name of the waveform, for name of the parent stimuli or measurement, for the name of all the variables occurring in the waveform. In short, the storage entries provide links which allow one to search the associative memory very fast.

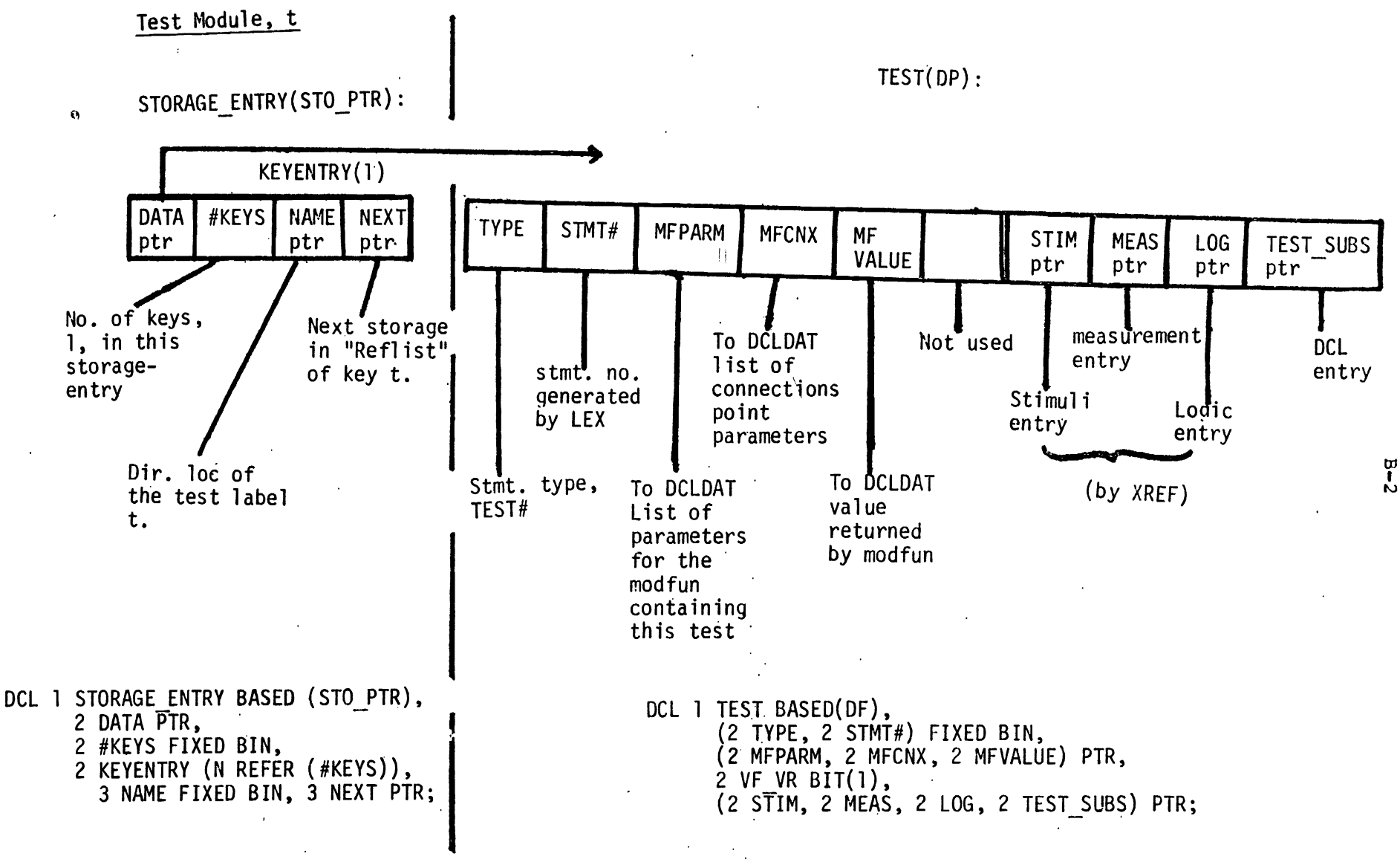
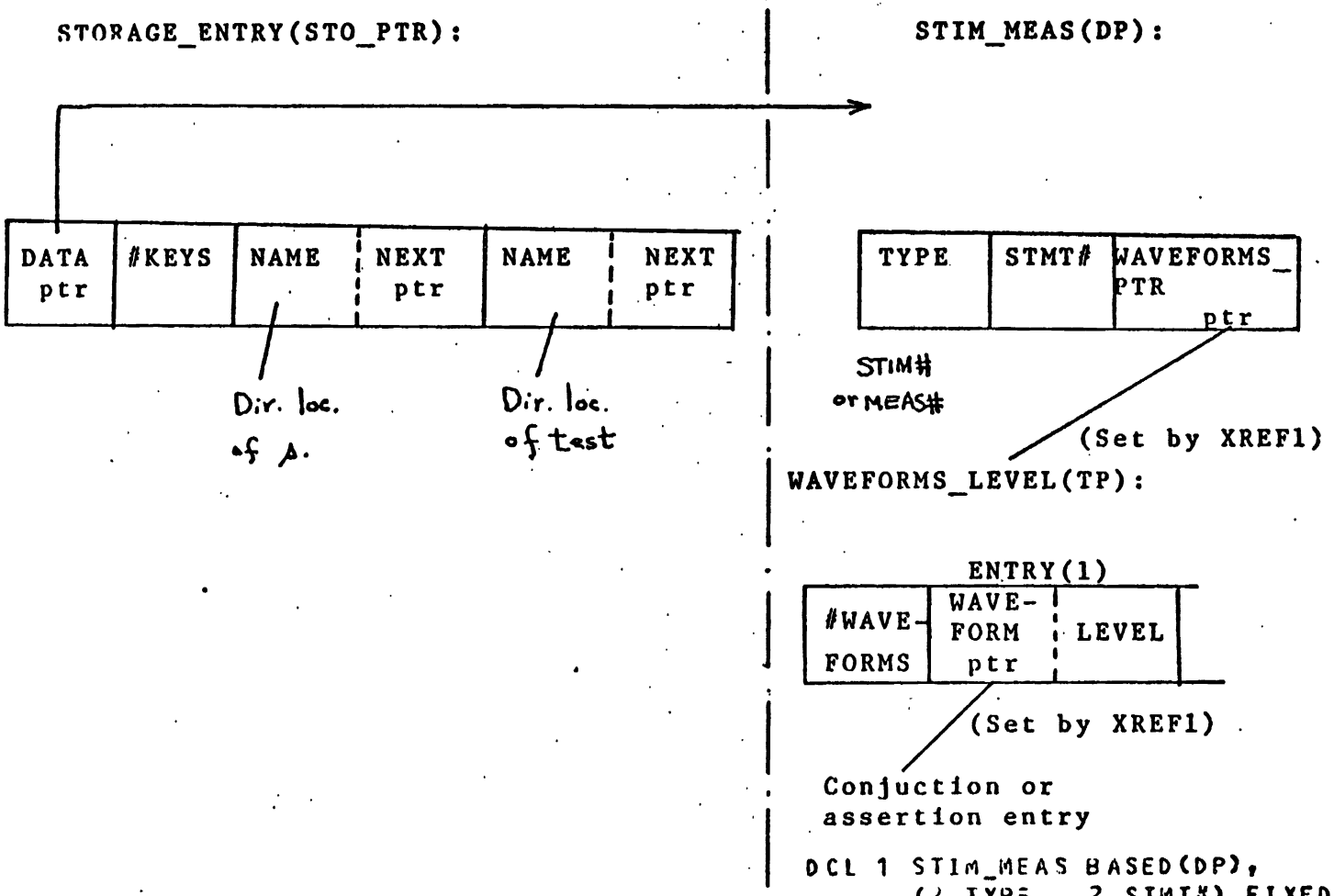


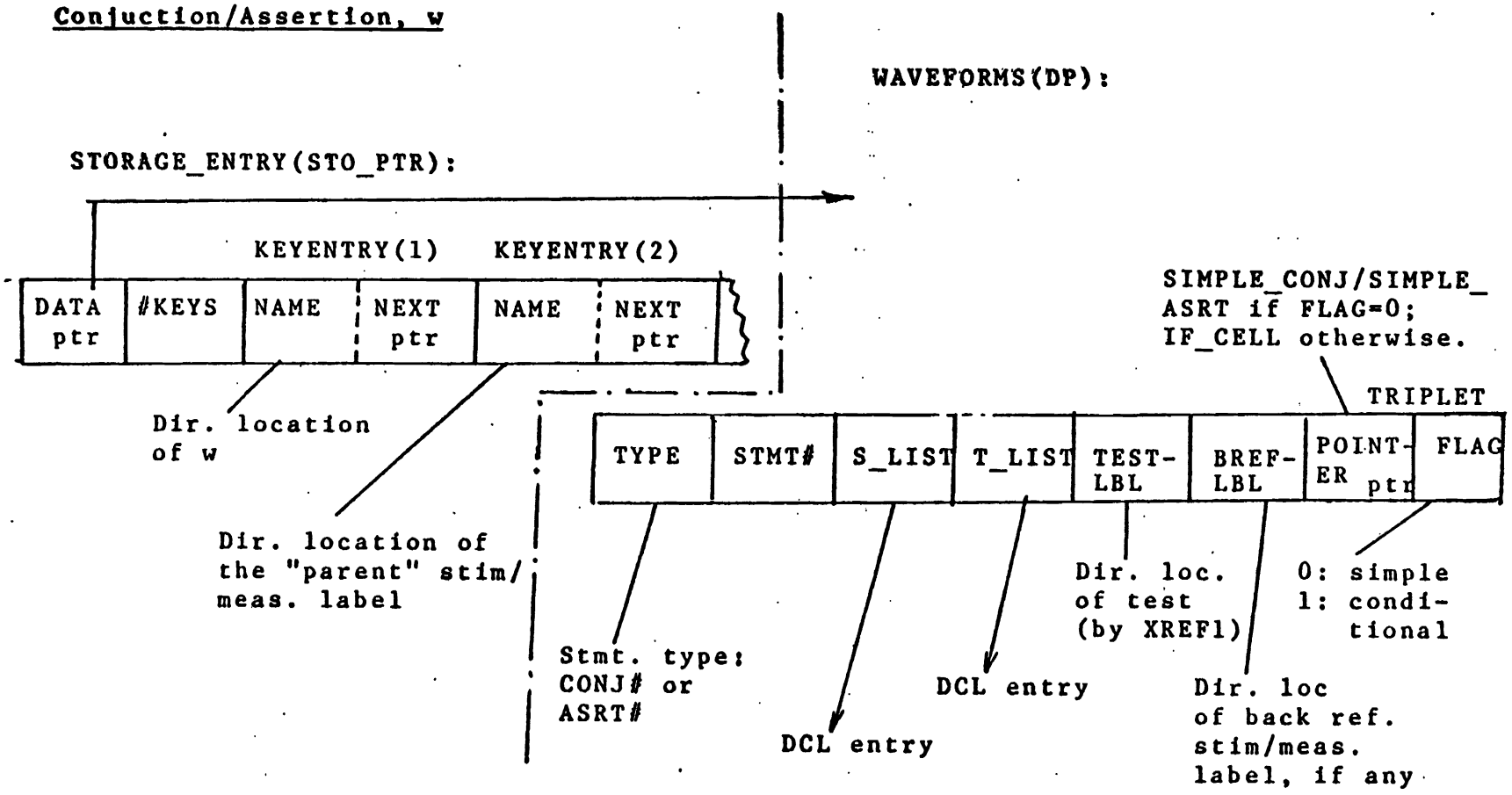
FIGURE B.1 TEST

Stimuli/measurement, s



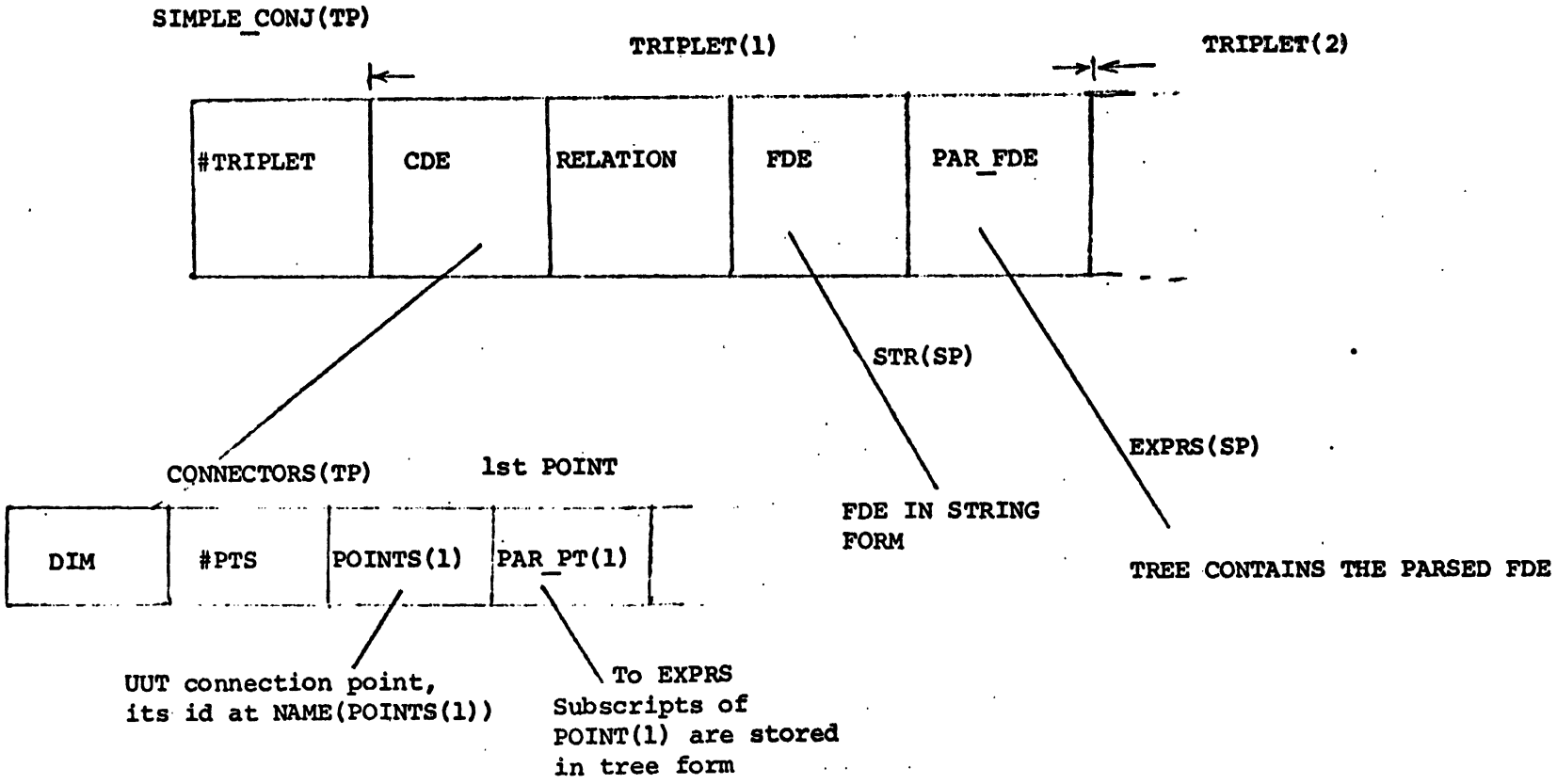

```
DCL 1 STIM_MEAS BASED(DP),
      (2 TYPE, 2 STMT#) FIXED BIN,
      2 WAVEFORMS_PTR  PTR;
DCL 1 WAVEFORMS_LEVEL BASED(TP),
      2 #WAVEFORMS FIXED BIN,
      2 ENTRY(N REFER(#WAVEFORMS)),
        3 WAVEFORM PTR,  3 LEVEL FIXED BIN;
```

FIGURE B.2: STIM_MEAS AND WAVEFORMS_LEVEL




```
DCL 1 WAVEFORMS ALIGNED BASED(DP),
      (2 TYPE,  2 STMT#) FIXED BIN,
      (2 S_LIST,  2 T_LIST) PTR,
      (2 TEST_LBL,  2 BREF_LBL) FIXED BIN,
      2 TRIPLET,
        3 POINTER PTR,  3 FLAG BIT(1);
```

FIGURE B.3: WAVEFORMS

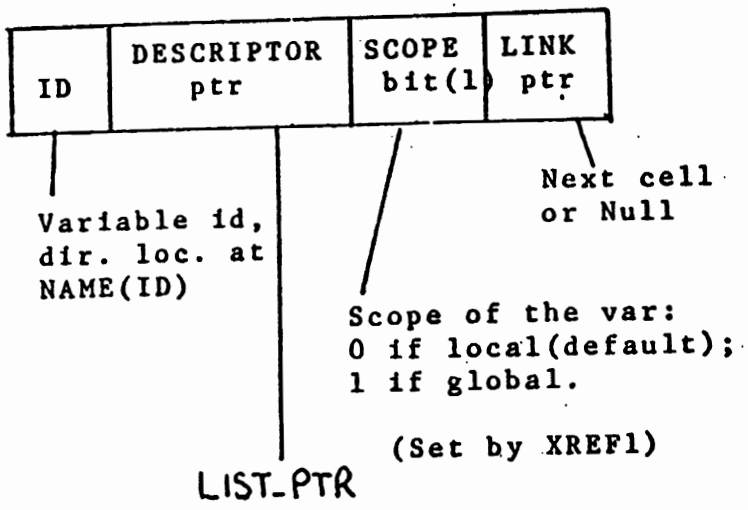



```
DCL 1 SIMPLE_CONJ BASED(TP),
    2 #TRIPLET FIXED BIN,
      2 TRIPLET( NO REFER(#TRIPLET)),
        3 CDE PTR,  3 RELATION CHAR(1),  3 FDE PTR,
        3 PAR_FDE PTR;
DCL 1 CONNECTORS BASED(TP),
      2 DIM CHAR(#LEN),
      2 #PTS FIXED BIN,
      2 P_POINTS (NP REFER(#PTS)),
        3 POINTS FIXED BIN,
        3 PAR_PT PTR;
```

FIGURE B.4: SIMPLE_CONJ AND CONNECTORS

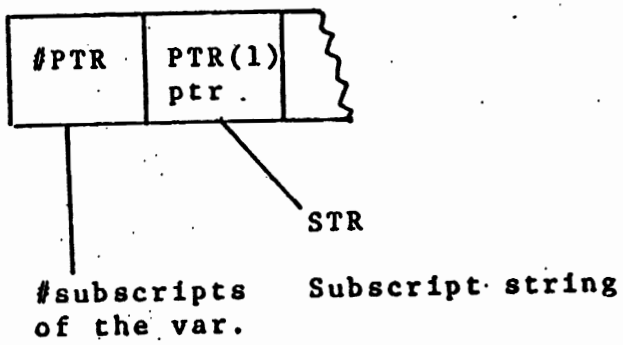
DECLARATION (DCL)

DCL(TP):



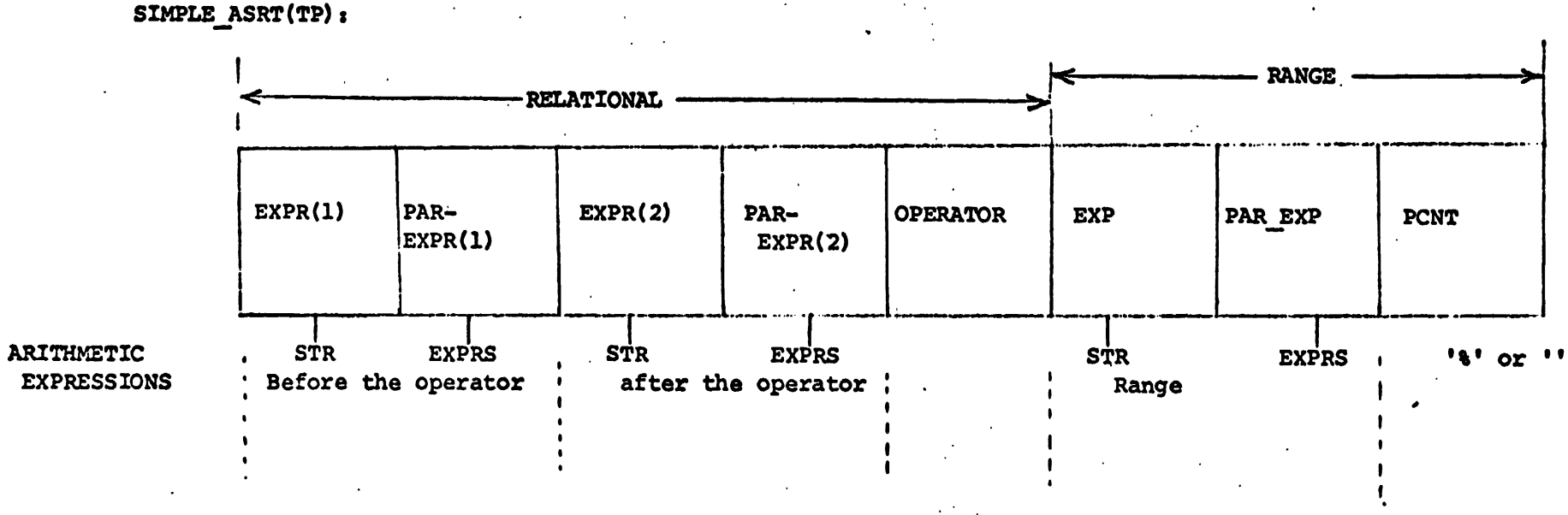

```
DCL 1 DCL ALIGNED BASED(TP),
      2 ID FIXED BIN,  2 DESCRIPTOR PTR,
        2 SCOPE BIT(1),  2 LINK PTR;
```

LIST_PTR(TP):




```
DCL 1 LIST_PTR BASED(TP),
      2 #PTR FIXED BIN,
      2 PTR(NC REFER(LIST_PTR.#PTR)) PTR;
```

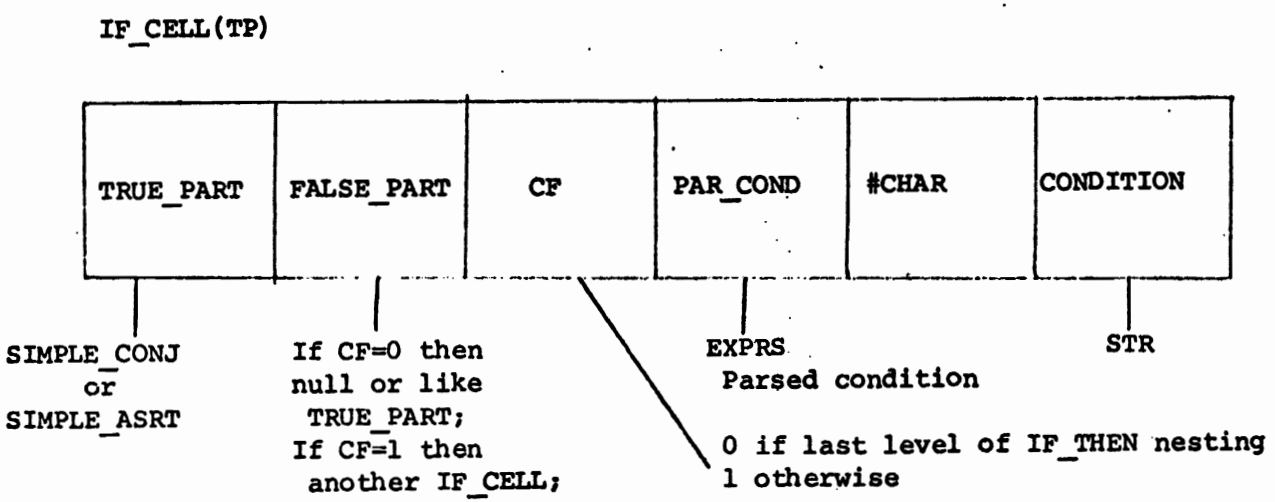
FIGURE B.5: DCL




```
DCL 1 SIMPLE_ASRT BASED(TP),
      2 RELATIONAL,
      3 EXPRE(2),
        4 EXPR PTR,
        4 PAR_EXPR PTR,
      3 OPERATOR CHAR(2),
      2 RANGE,
      3 EXP PTR,
      3 PAR_EXP PTR,
      3 PCNT CHAR(1);
```

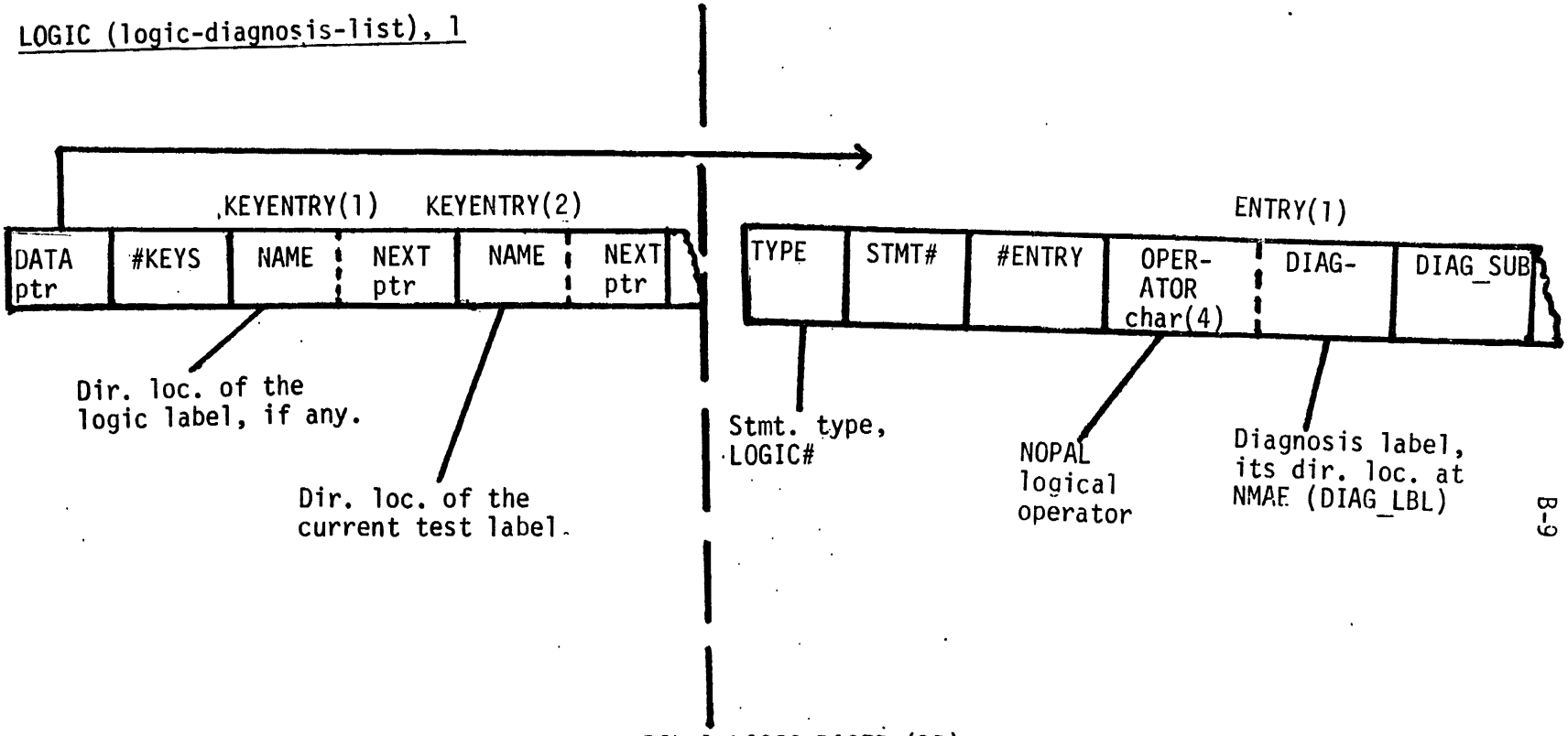
Note: The record STR contains the arithmetic expression in the form of character string. Structure EXPRS is the tree structure containing the parsed arithmetic expression.

FIGURE B.6: SIMPLE_ASRT




```
DCL 1 IF_CELL ALIGNED BASED(TP),
      (2 TRUE_PART, 2 FALSE_PART) PTR,
       2 CF BIT(1), 2 PAR_COND PTR, 2 #CHAR FIXED BIN,
       2 CONDITION CHAR(LS REFER(IF_CELL.#CHAR));
```

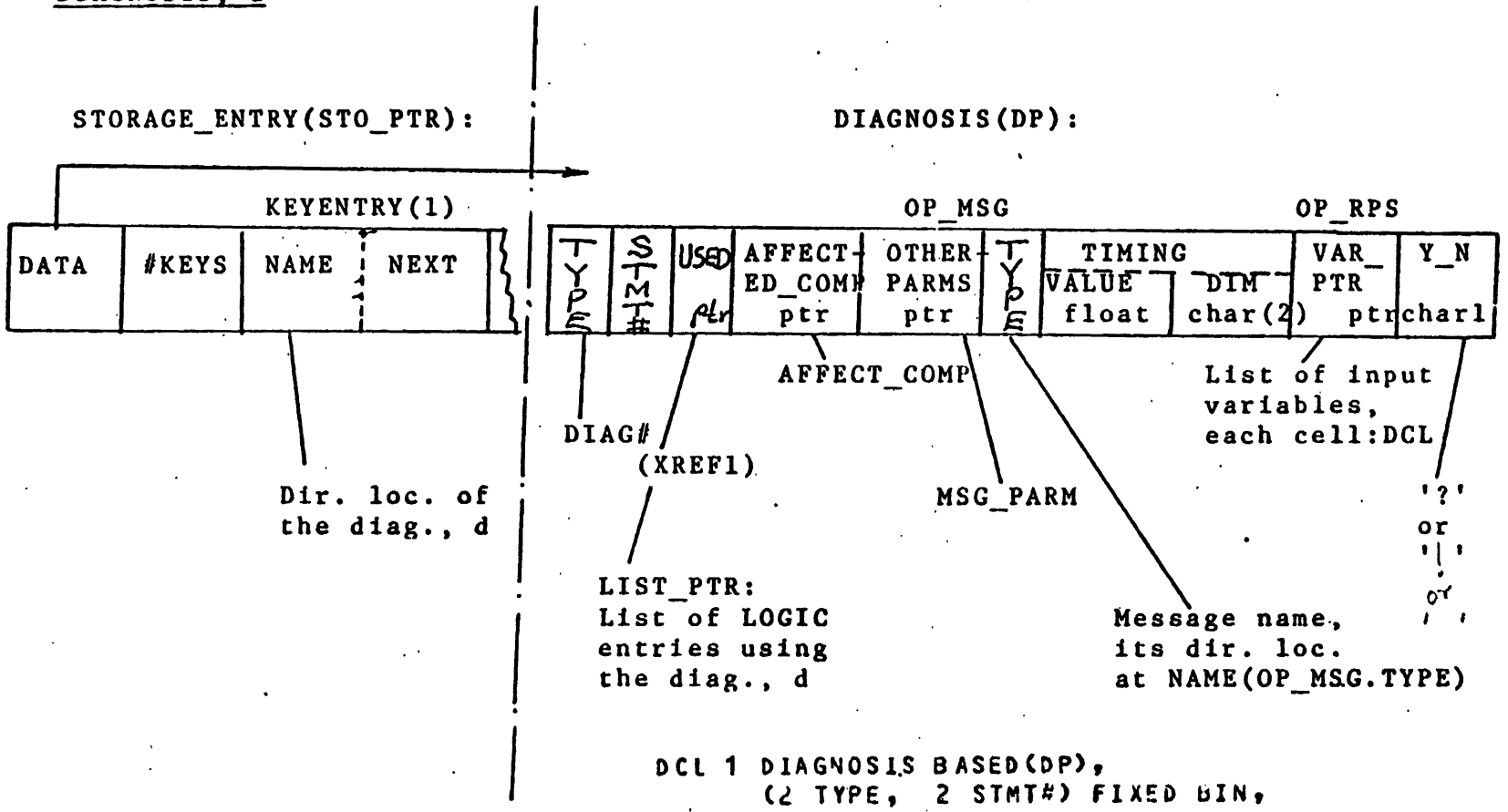
FIGURE B.7: IF_CELL




```
DCL 1 LOGIC BASED (DP),
      (2 TYPE, 2 STMT#) FIXED BIN,
      2 #ENTRY FIXED BIN,
      2 ENTRY (NO REFER(#ENTRY)),
        3 OPERATOR CHAR(4), 3 DIAG_LBL FIXED BIN,
        3 DIAG_SUBS PTR;
```

FIGURE B.8 LOGIC

DIAGNOSIS, d

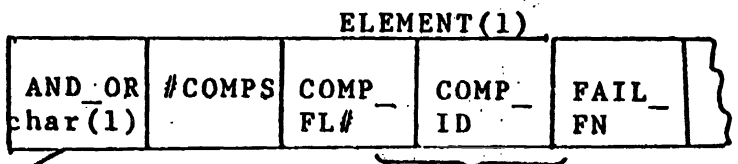



```
DCL 1 DIAGNOSIS BASED(DP),
      (2 TYPE,  2 STMT#) FIXED BIN,
      2 USED PTR,
      2 OP_MSG,
        (3 AFFECTED_COMP,  3 OTHER_PARMS) PTR,
        3 TYPE FIXED BIN,
        3 TIMING,
          4 VALUE DEC FLOAT,  4 DIM CHAR(@LEN),
      2 OP_RPS,
        3 VAR_PTR PTR,  3 Y_N CHAR(1);
```

FIGURE B.9 DIAGNOSIS

AFFECTED COMPONENTS

AFFECT_COMP(TP);



Affected component:

AND('&')
or
OR('|')

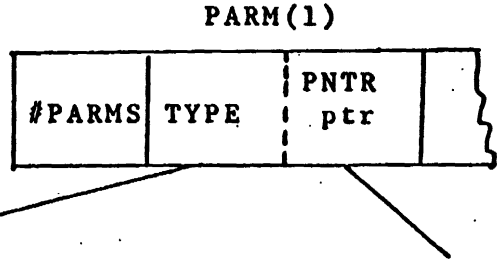
Initially, either COMP_ID and FAIL_FN are given only, or COMP_FL# is given only.
Then after XREF1, they point to the corresponding directory locations of the component-failure-seq# component id, and failure function, respectively.

```
DCL 1 AFFECT_COMP BASED(TP),
      2 AND_OR CHAR(1),  2 #COMPS FIXED BIN,
      2 ELEMENT(NC REFER(#COMPS)),
        (3 COMP_FL#,  3 COMP_ID,  3 FAIL_FN) FIXED BIN;
```

FIGURE B.10 AFFECT_COMP

OP-MSG PARAMETERS

MSG_PARM(TP);



Type of parameter:
$A (=2), variable;
$D (=3), number ;
$BIT (=4), Bit string;
$CHAR (=5), Char. string.

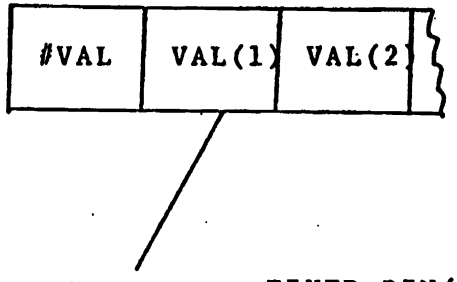
STR, if TYPE≠$A;
DCL, if TYPE=$A.

```
DCL 1 MSG_PARM BASED(TP),
      2 #PARMS FIXED BIN,
      2 PARM(NC REFER(MSG_PARM.#PARMS)),
        3 TYPE FIXED BIN,  3 PNTR PTR;
```

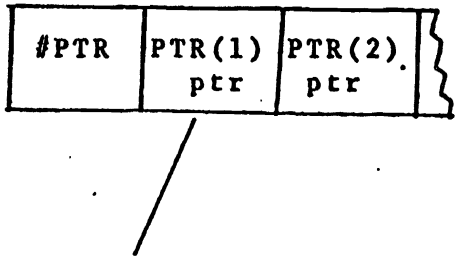
FIGURE B.11: MSG_PARM

LIST-VAL/LIST-PTR (list of values or pointers)

LIST_VAL(TP):



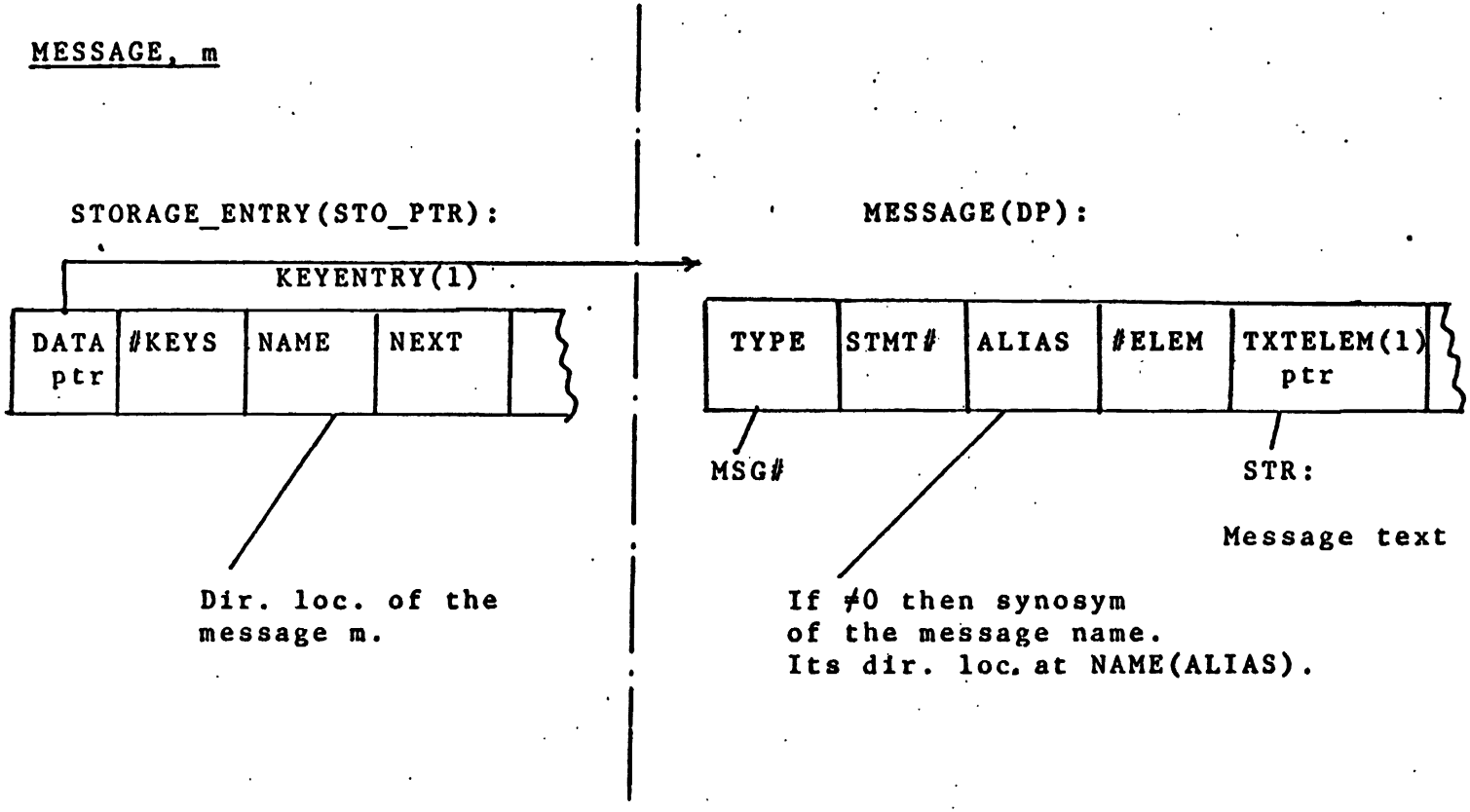
LIST_PTR(TP):




```
DCL 1 LIST_VAL BASED(TP),
      2 #VAL FIXED BIN,
      2 VAL(NC REFER(LIST_VAL.#VAL)) FIXED BIN;
```

```
DCL 1 LIST_PTR BASED(SP),
      2 #PTR FIXED BIN,
      2 PTR(NF REFER(LIST_PTR.#PTR)) PTR;
```

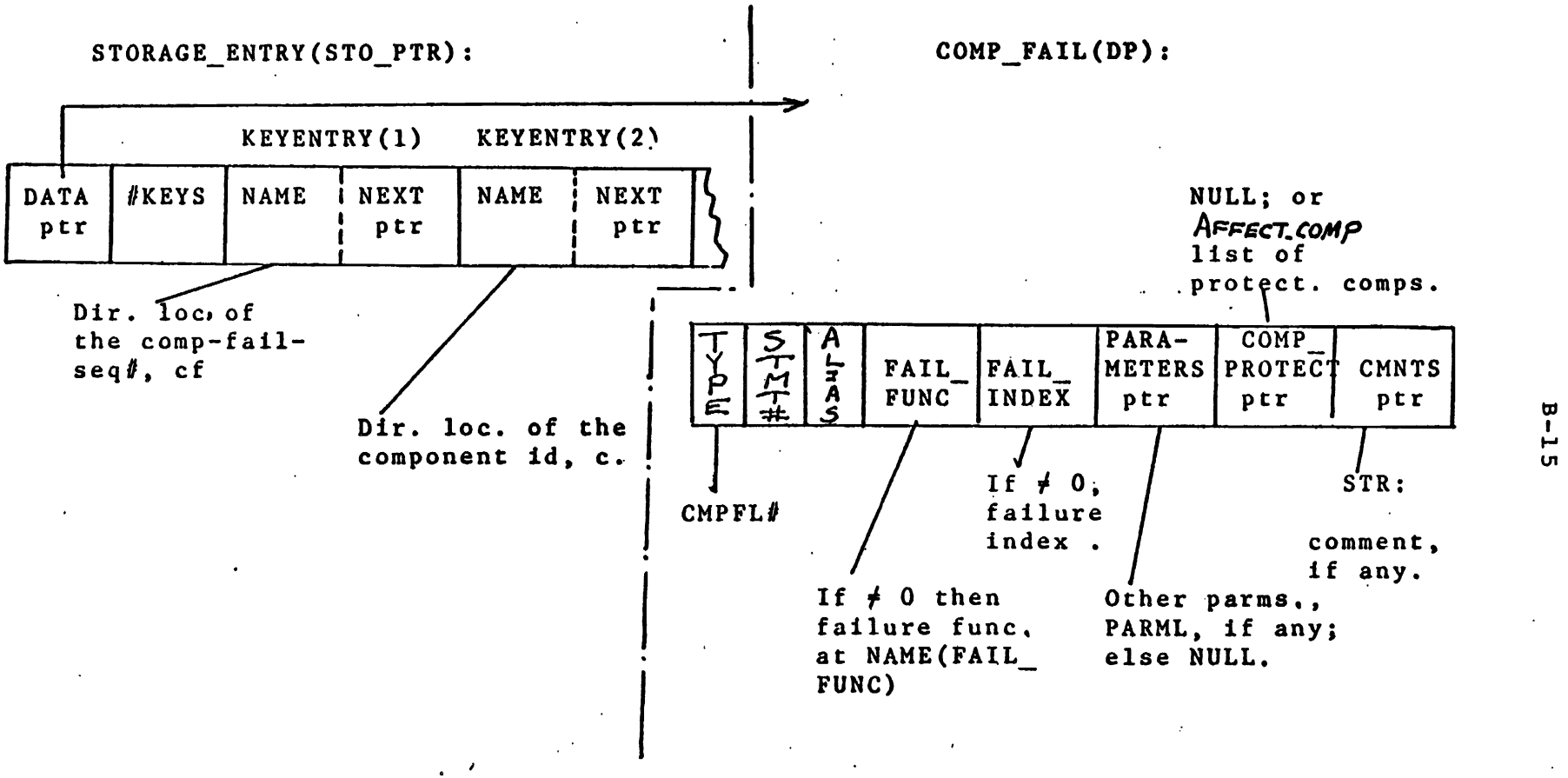
FIGURE B.12: LIST_VAL AND LIST_PTR




```
DCL 1 MESSAGE BASED(DP),
      (2 TYPE,  2 STMT#) FIXED BIN,
      2 MSG_SYNONYM FIXED BIN ,
      2 #ELEMS FIXED BIN,
      2 TXTELEMS(NP REFER(#ELEMS)) PTR;
```

FIGURE B.13: MESSAGE

COMPONENT FAILURE, cf: f(c)

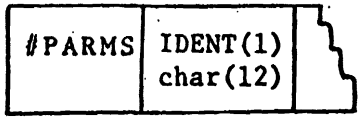



```
DCL 1 COMP_FAIL BASED(DP),
      (2 TYPE,  2 STMT#) FIXED BIN,
      (2 COMP_SYNONYM,  2 FAIL_FUNC,  2 FAIL_INDEX) FIXED BIN,
      (2 PARAMETERS,  2 COMP_PROTECT,  2 CMNTS) PTR;
```

FIGURE B.14: COMP_FAIL

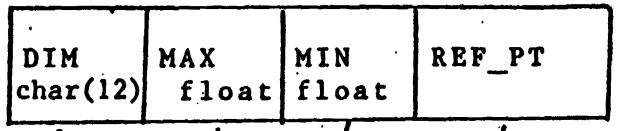
PARAMETER LIST/PROTECTIVE LIMITS

PARAMETER LIST/
PARML(TP):



Parameter name

LIMIT(TP):



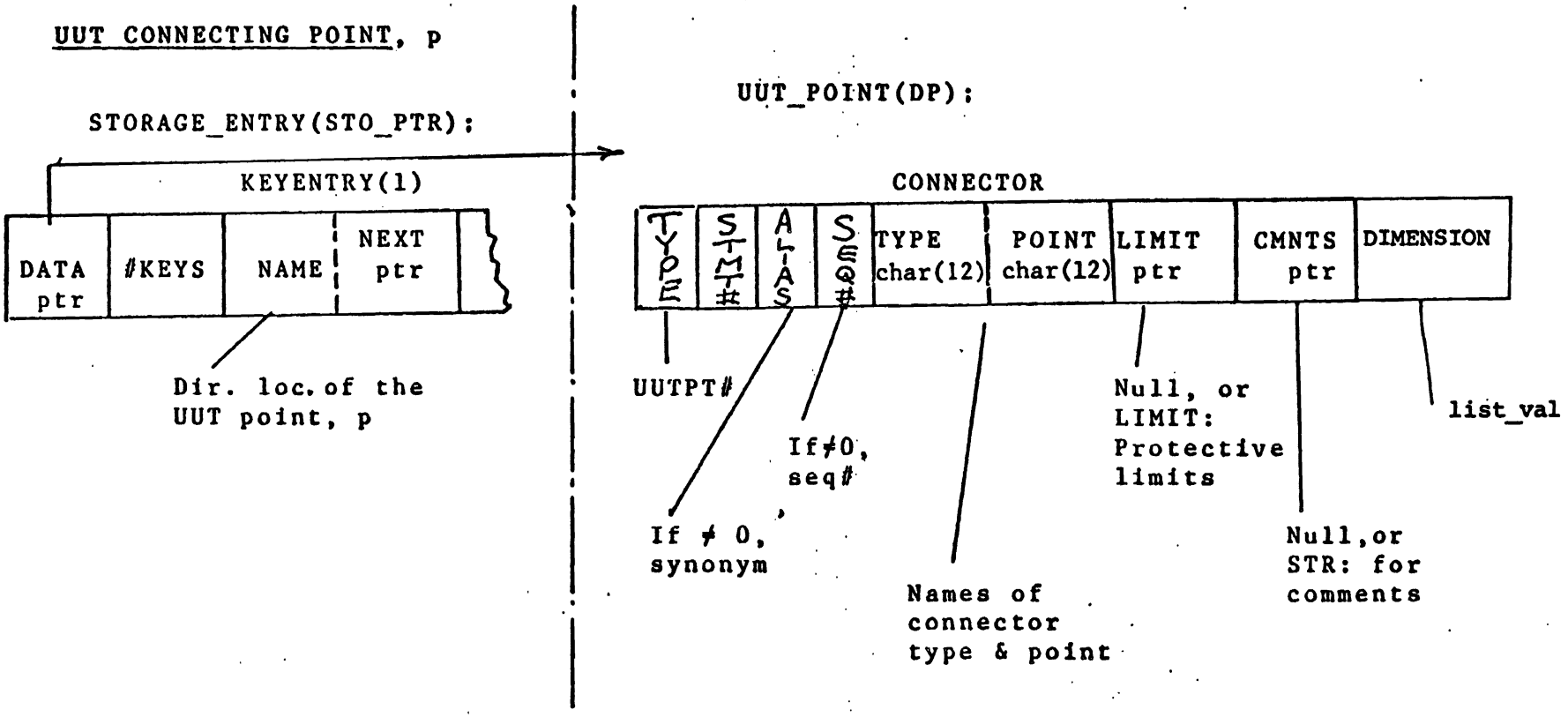
Dimension,
if any

Upper/lower
limits, if any

Reference point:
If ≠ 0, then its
dir. loc. at NAME(REF_PT)

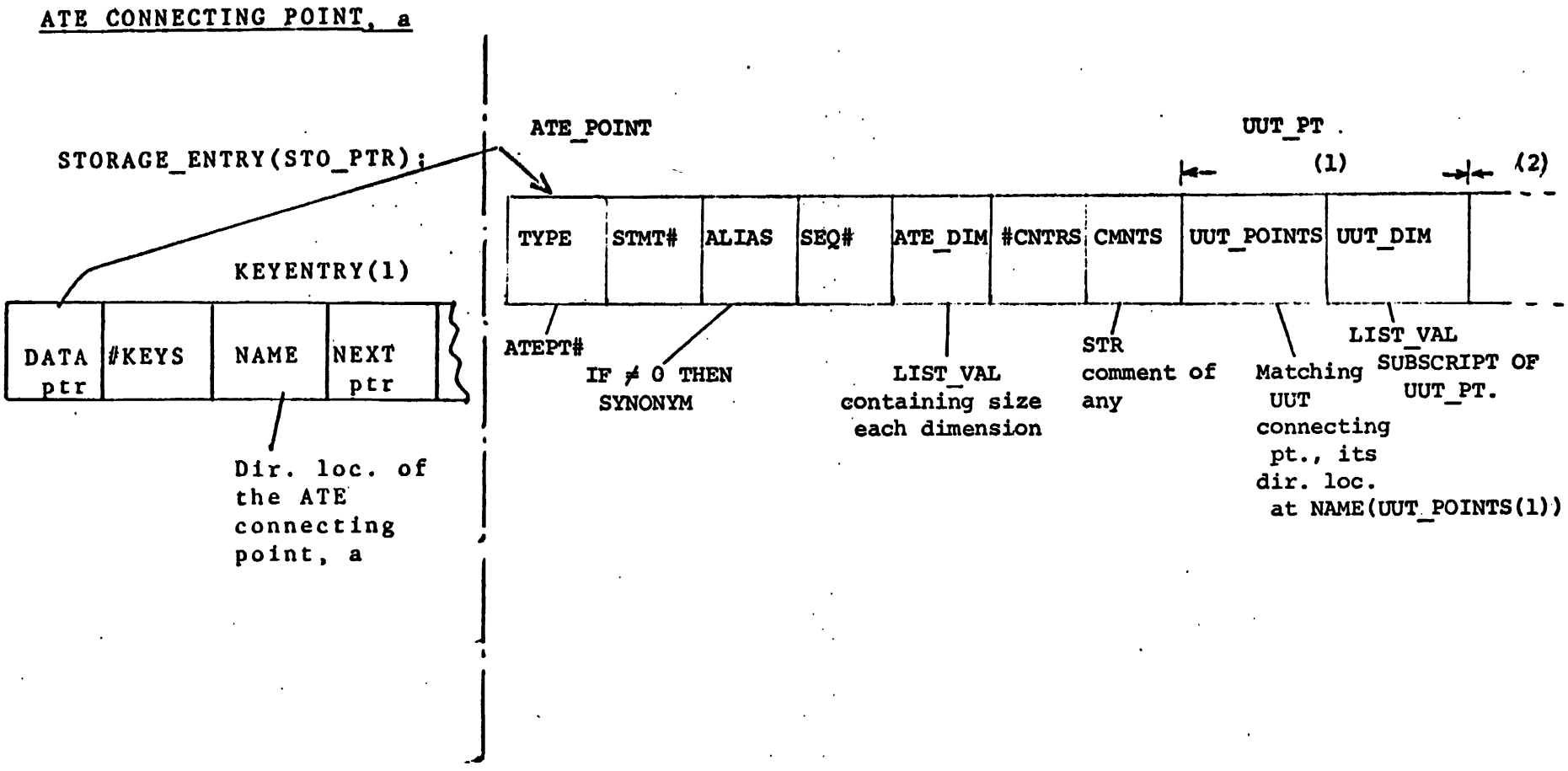
```
DCL 1 LIMIT BASED(TP),
      2 DIM CHAR(8LEN),
      (2 MAX,  2 MIN) DEC FLOAT,
      2 REF_PT FIXED BIN;
```

FIGURE B.15: LIMIT




```
DCL 1 UUT_POINT BASED(DP),
      (2 TYPE,  2 STMT#) FIXED BIN,
      (2 PIN_SYNONYM,  2 ENTRY_SEQ#) FIXED BIN,
      2 CONNECTOR,
        (3 TYPE,  3 POINT) CHAR(@LEN),
      (2 LIMIT,  2 CMNTS, 2 DIMENSION) PTR;
```

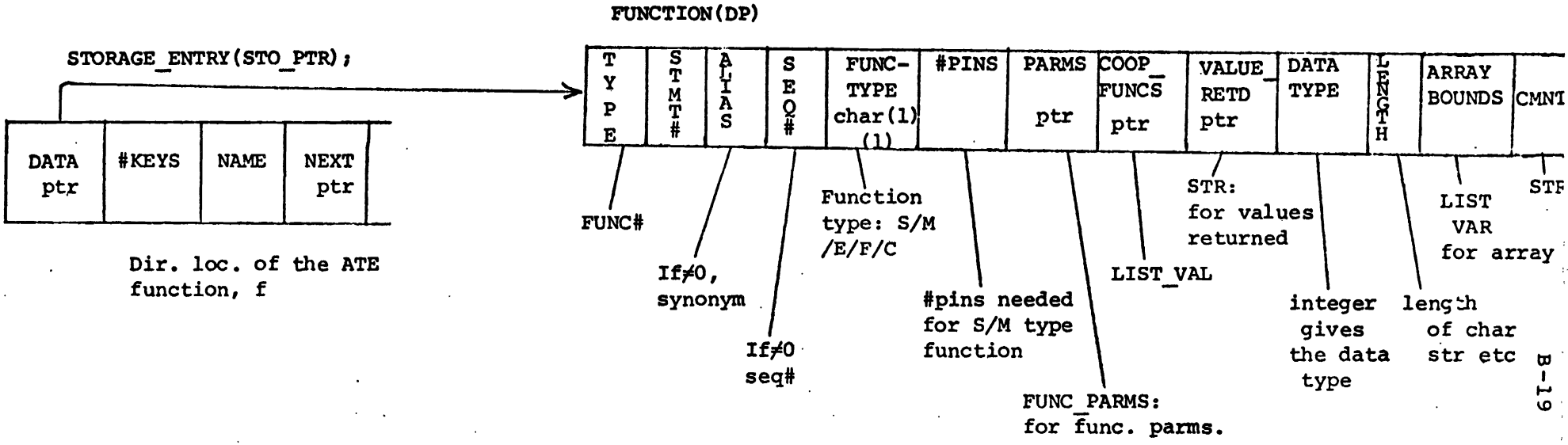
FIGURE B.16 UUT_POINTS




```
DCL 1 ATE_POINT BASED(DP),
      (2 TYPE,  2 STMT#) FIXED BIN,
      (2 POINT_SYNONYM,  2 ENTRY_SEQ#) FIXED BIN,
      2 ATE_DIM PTR,
      2 #CNTRS FIXED BIN,   2 CMNTS PTR,
      2 UUT_PT(NC REFER(#CNTRS)),
        3 UUT_POINTS FIXED BIN,
        3 UUT_DIM PTR;
```

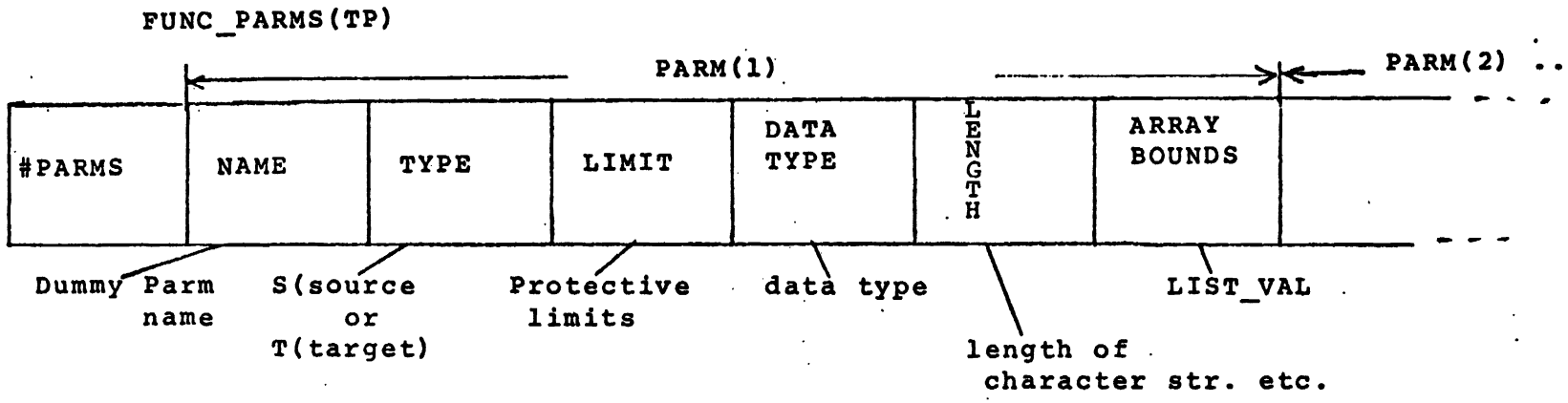
FIGURE B.17 ATE_POINT

ATE FUNCTION, F




```
DCL 1  FUNCTION BASED(DP),
       (2 TYPE, 2 STMT#) FIXED BIN,
       (2 FUNC_SYNONYM, 2 ENTRY_SEQ#) FIXED BIN,
       2 FUNC_TYPE CHAR(1), 2 #PINS FIXED BIN,
       (2 PARMS,  2 COOP_FUNCS) PTR,
        2 VALUE_RETD  PTR,
        2 DATA_TYPE FIXED BIN,
        2 LENGTH FIXED BIN,
        2 ARRAY_BOUNDS PTR,
        2 CMNTS PTR;
```

FIGURE B.18: FUNCTION



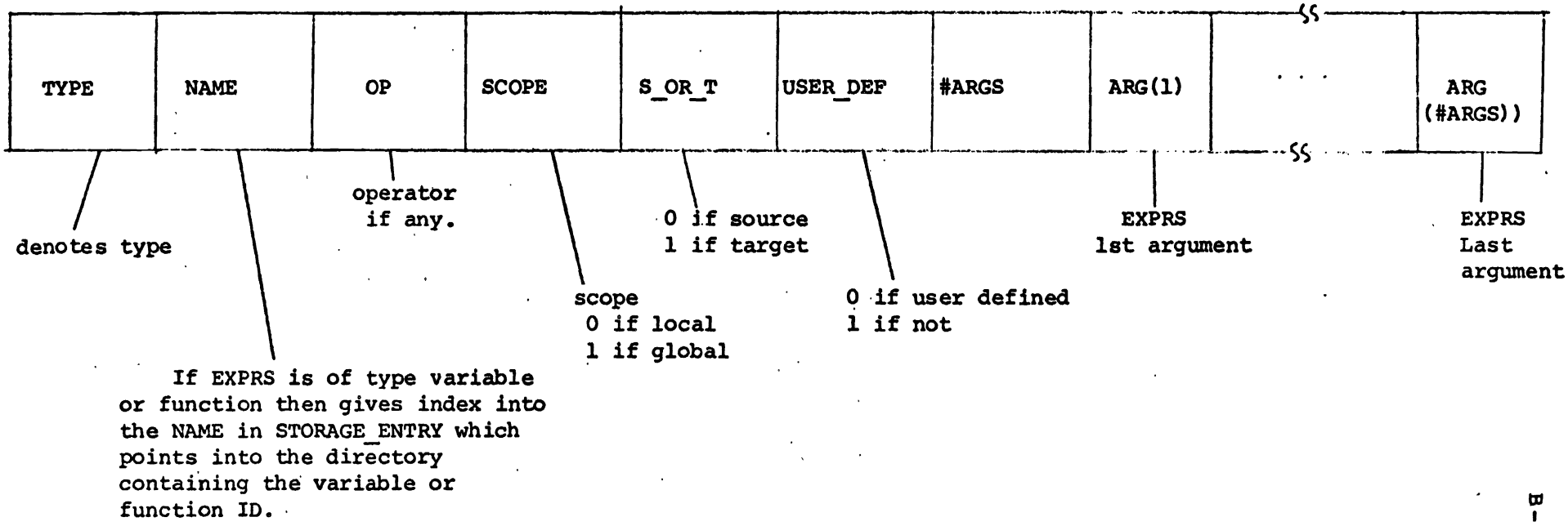

```
1  FUNC_PARMS  BASED(TP),
   2  #PARMS  FIXED BIN,
   2  PARM(NV REFER (FUNC_PARMS.#PARMS)),
      3  NAME CHAR(12),
      3  TYPE CHAR(1),
      3  LIMIT PTR,
      3  DATA_TYPE  FIXED BIN,
      3  LENGTH FIXED BIN,
      3  ARRAY-BOUNDS PTR;
```

Data type:

data type	integer stored in DATA_TYPE
Real	1
Integer	2
Char	3
bit	4
Octal	5
Hex	6
Complex	7

FIGURE B.19: FUNC_PARMS

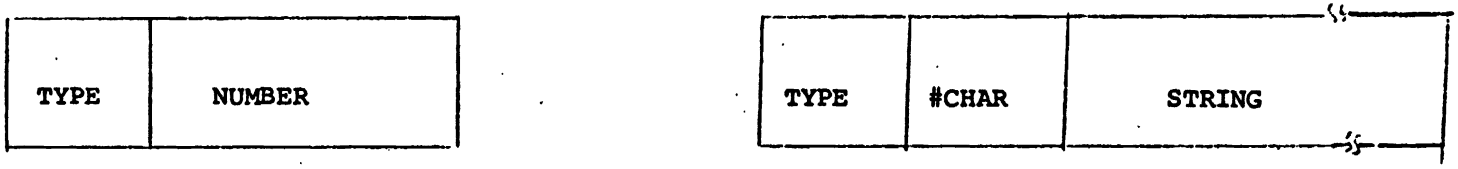
EXPRS




```
DCL 1 EXPRS BASED(SP),
      2 TYPE FIXED BIN,
      2 NAME FIXED BIN,
      2 OP CHAR(2),
      2 SCOPE BIT(1),
      2 S_OR_T BIN FIXED,
      2 USER_DEF BIT(1),
      2 #ARGS FIXED BIN,
      2 ARG(N REFER(#ARGS)) PTR;
```

FIGURE B.20: EXPRS

LEAFS




```
DCL 1 LEAFX BASED(VP),
      2 TYPE FIXED BIN,
      2 NUMBER FIXED BIN;
DCL 1 LEAFL BASED(VP),
      2 TYPE FIXED BIN,
      2 NUMBER DEC FLOAT;
DCL 1 LEAFCHAR BASED(VP),
      2 TYPE FIXED BIN,
      2 #CHAR FIXED BIN,
      2 STRING CHAR(LS REFER(LEAFCHAR.#CHAR));
DCL 1 LEAFBIT BASED(VP),
      2 TYPE FIXED BIN,
      2 #BIT FIXED BIN,
      2 BITSTR BIT(LS REFER(#BIT));
```

FIGURE B.21 LEAFS

Type of EXPRS or LEAF

TYPE	Mnemonic	description
3	$FX	fixed bin
4	$BIT	bit string
5	$CHAR	char string
6	$FL	floating point
7	$STAR	*was found (default value for arg.)
8	$ARRAY	array variable
11	$BORAR	Function (arith. or boolean or otherwise or array or var)
12	$ARITH	arith fn or array or var
13	$BOOL	bool. fn or array or var
14	$MSFN	Meas. or stimuli fn.
15	$CNXPT	connection point
21	$AROP	arithmetic operator
22	$BOP	boolean operator
23	$BREL	boolean predicate or relation
31	$VDE	value dim expr.
35	$ASRT	to store an assertion which occurs in the parameter of the function call

FIGURE B.22: TYPES OF EXPRS OR LEAF

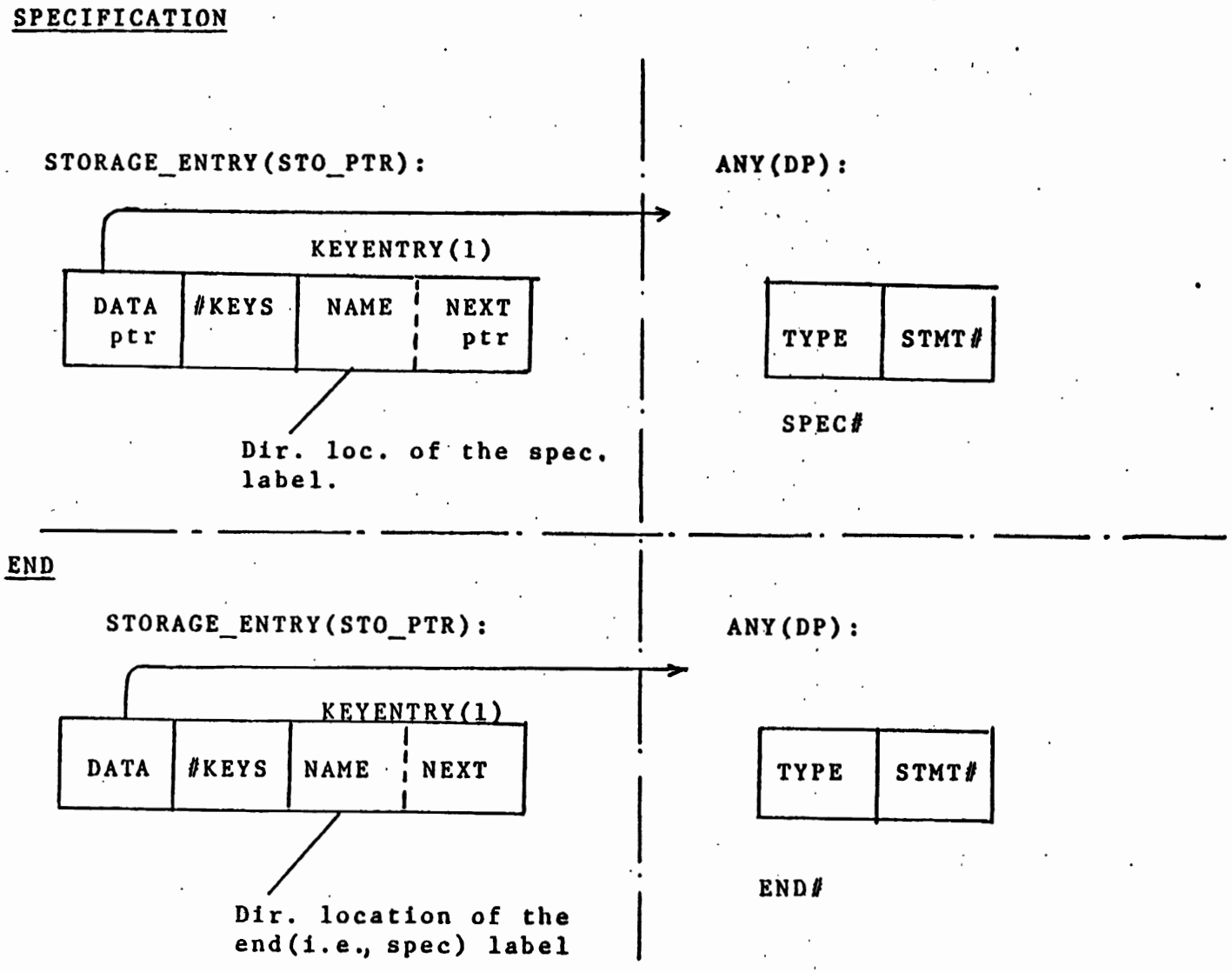


FIGURE B.23: ANY

STORAGE_ENTRY (STO_PTR)

DATA	#KEYS	NAME	NEXT

Directing location of system generated. name for dataspec.

DATASPEC(DP)

TYPE	STMT#	DCLIST

To DCLDAT. List of variables or the structure declared by this statement

FIGURE B.24: DATASPEC

DCLDAT(TP)

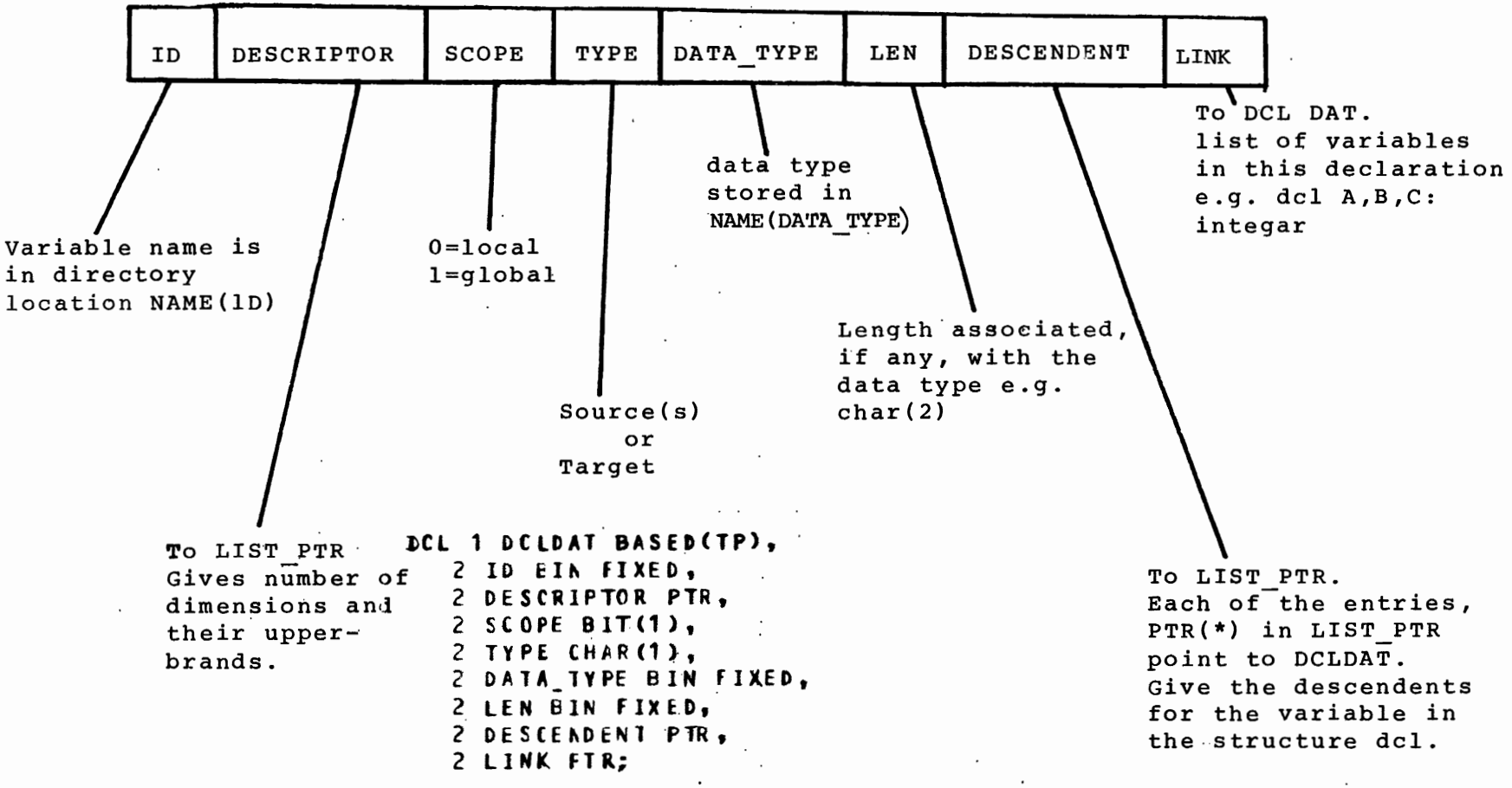


FIG B.25: DCLDAT

```
TEST 5;
   STIMULI  3(5);
      CONJUNCTION
      (<J24_B,GND> = CONST_5(27.5 VOLT))&
      (<J16> = SIGNAL_AM(2.001 MHZ, -95DB, Y,
                                    1 KHZ))  SOURCE:  Y;
   ASSERTION:
      IF X > 0  THEN   A = B + SIN(C);
                  TARGET: A
                  SOURCE: X,B,C
LOGIC:
     |D2,  & D3
```

Legend: (Refer to the diagram)

1(a) Any structure with '...' in it indicates that some fields have been omitted. Only the important info. is filled in.
Ex: TRIPLET(2) in SIMPLE_CONJ; DIRCTRY.

(b) Any field with '.' in it indicates that information has not been filled in the figure.

2 denotes null pointer : 

3 STMT# has not been filled in any of the structures.

4 --> used to show index into (i.e. pointing to) the storage entry or dirctry. All dotted pointers point to dirctry unless otherwise indicated.

5 Used to show tree structure without showing the details of the node: 

FIGURE B.26 AN EXAMPLE SHOWING THE ASSOCIATIVE MEMORY FOR THE TEST MODULE SPECIFICATION GIVEN ABOVE.

DIRCTRY

INDEX	KEYNAME	KEYTYPE ...
1	5	TEST#
2	3	STIM#
3	GND	UUTPT#
4	J24_B	UUTPT#
5	CONST_S	STIM#
6	J16	UUTPT#
7	Y	VAR#
8	SIGNAL_AM	STIM# ...
9	X	VAR#
10	A	VAR#
11	B	VAR#
12	SIN	FUNC#
13	C	VAR#
14	D2	DIAG#
15	D3	DIAG# ...

FIGURE B.26: (continued)

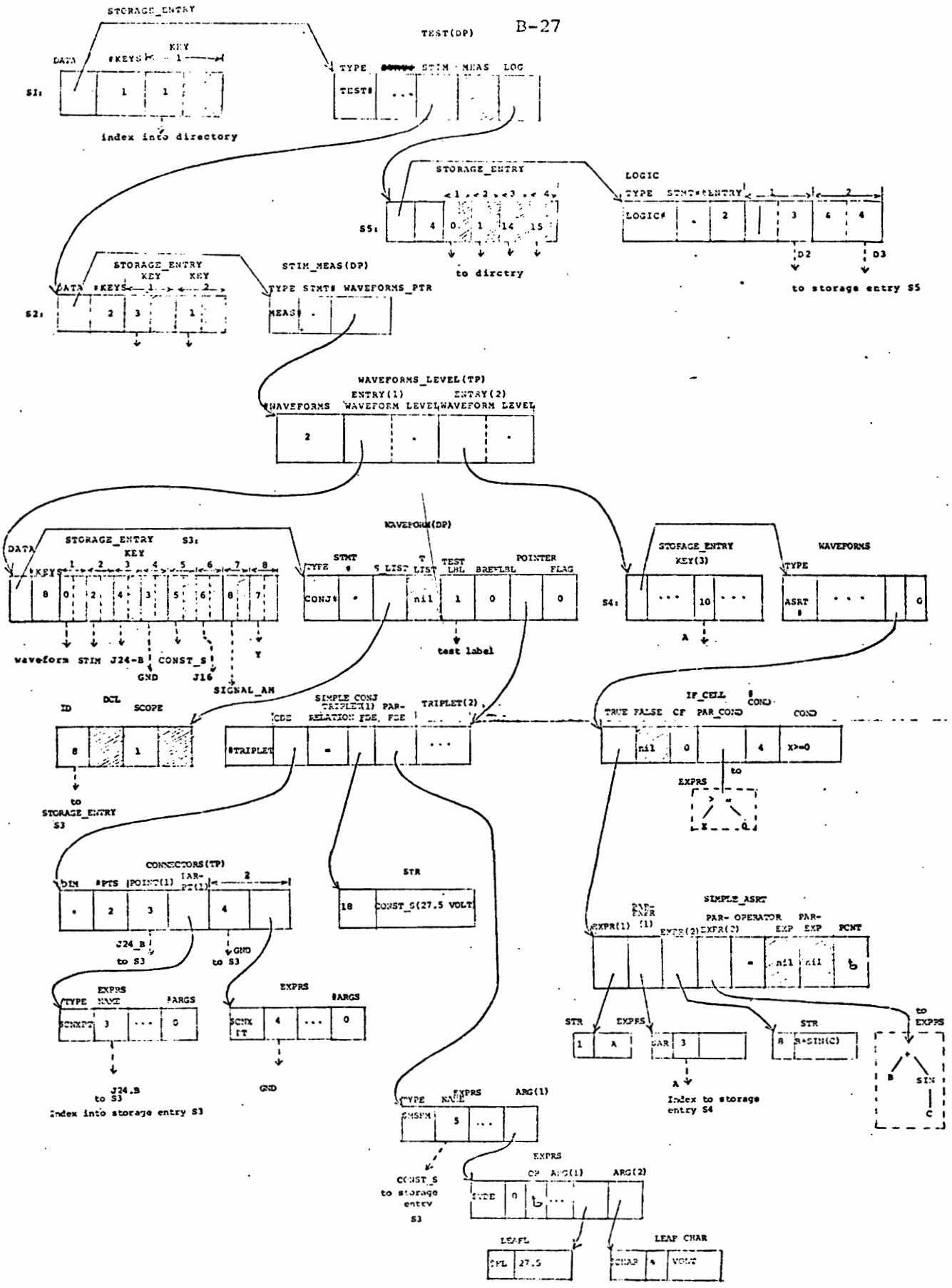

STORAGE_ENTRY
DATA
#KEYS
KEY
S1:
index into directory
TEST(DP)
TYPE
STIM
MEAS
LOG
TEST#
STORAGE_ENTRY
S5:
to directry
LOGIC
LOGIC#
D2
D3
to storage entry S5
S2:
STIM_MEAS(DP)
TYPE STMT# WAVEFORMS_PTR
MEAS#
WAVEFORMS_LEVEL(TP)
ENTRY(1)
ENTRY(2)
#WAVEFORMS
WAVEFORM LEVEL
S3:
WAVEFORM(DP)
STMT #
TEST LBL
BREF LBL
POINTER
FLAG
CONJ#
nil
waveform
STIM
J24-B
CONST_S
J16
GND
SIGNAL_AM
test label
S4:
STORAGE_ENTRY KEY(3)
WAVEFORMS
ASRT
ID
DCL
SCOPE
to STORAGE_ENTRY S3
SIMPLE_CONJ
TRIPLET(1)
TRIPLET(2)
RELATION
#TRIPLET
IF_CELL
TRUE
FALSE
CF
PAR_COND
COND
nil
X>=0
to EXPRS
CONNECTORS(TP)
BIN
#PTS
POINT(1)
PAR-PT(1)
J24_B to S3
GND to S3
EXPRS
TYPE
NAME
#ARGS
CONNPT
J24.B to S3
Index into storage entry S3
STR
18
CONST_S(27.5 VOLT)
SIMPLE_ASRT
EXPR(1)
EXPR(2)
PAR-OPERATOR
PAR-EXP
PCNT
STR
EXPRS
VAR
A
Index to storage entry S4
B+SIN(C)
B
SIN
C
ARG(1)
ARG(2)
CONST_S to storage entry S3
LEAF
LEAF CHAR
CHAR
VOLT

FIGURE B.26

DESCRIPTION OF THE AUTOMATIC PROGRAM GENERATION PROJECT

Department of Computer and Information Science
University of Pennsylvania
August 1982

1. Introduction.

The Automatic Program Generation Project has been engaged since 1975 in developing very-high-level programming languages. The emphasis has been on languages which are purely non-procedural. However other aspects for attaining the very high-level effect were included as well. By non-procedural, it is meant that statements in the language can be in an arbitrary order and are purely descriptive. The statement describes only organization of the databases and equations. There are no control facilities such as for iterations, input/output or memory allocations.

These language processors must incorporate built-in intelligence in mathematics and computer programming in order to be able to translate the very-high-level non-procedural language input into a conventional program. The approach of the project has been to progressively develop more powerful processors and test the inherent concepts in various application areas. The general requirements of very-high-level programming languages are described in the next section. The remaining sections describe specific developments and applications. Two processors have been developed to date: MODEL for general-purpose computations, and NOPAL for automatic testing of analog systems and assemblies. Research on MODEL also included considerations of impact of new computer architectures, especially data-flow machines.

The research has been supported by various agencies. The basic research on MODEL was supported by the Office of Naval Research. The extensions of MODEL for linking sub-systems using distributed processing and the application to econometric modelling was supported by the National Science Foundation. The application of MODEL in the business data processing was supported by the Internal Revenuue Service. The NOPAL system development was supported by the U.S. Army.

2. Very-High-Level Programming Language Requirements.

Mathematical and computer complexities have been the main impediments in the development of computerized systems. Existing software systems are inadequate for computation development, especially for large-scale computation. The focus of the project has been on three classes of requirements.

A. Reduce the requisite expertise and labor of developers in use of numerical and statistical methods and in programming,

B. Facilitate testing and debugging of individual and multi-module systems and the experimentation with large-scale computational systems,

C. Facilitate growth of computational systems.

The boundary between the functions of the developer and the computer is thus delineated. The human developers are to be entirely responsible for the definition of the databases and formulation of equations, which is where developers make the real intellectual contributions. The intelligent system is responsible for interaction with the users, selection and use of solution methods and for programming and computation. State-of-the-art numerical methods are incorporated in the computations. The system must be expandable to include additional methods as they are developed. The above three areas are briefly described below:

A. Reducing Requisite Expertise of System Developers.

It is necessary to reduce the demands upon the developers for expertise in the areas of:

Mathematics--through automatic checking of completeness, non-ambiguity, and consistency of equational systems and automatic incorporation of appropriate statistical and numerical methods.

Computer Programming--through automatic generation of efficient computer programs for performing the computation of local and global systems, based on very-high-level specifications of databases and equations. The generated programs may be run on a variety of computer arcitectures, either locally or on a distributed basis, where local systems are executed in the computers of the respective developers which are linked through communications.

Experimenting with large-scale systems requires an enormous number of computations. Thus, efficiency of the computations is a key consideration in the automatic selection of statistical and numerical methods and in program design.

B. Facilitating Testing.

The development and testing is conducted first on a stand-alone local sub-system, followed by linkage of the local sub-systems (possibly in remote computers). Corrections to individual specifications must be made also in the very-high-level language. Debugging is facilitated by the fact that a variable in the non-procedural language can have only a single value. The language processor must also conduct checking of consistency and completeness throughout the specification. Several groups may participate in a development, linking various versions of their local sub-systems with other local sub-systems. Since, typically, the expertise resides in the local developing groups, it is very important to conduct computation of the stand-alone local systems and of the local portions of a global system in the respective local computers. The objective is to be able to update a global system in a matter of days after modifications have been made in any of the participating local systems. Another advantage is the distribution of the total effort of developing a large global system among a number of groups that participate in the cooperative computation. Parallel computation of local systems within the global system is also important as it reduces the computing time for experimenting with very large systems. Development and experimentation typically proceed in parallel.

C. Facilitating Growth of Systems.

The general tendency for systems to grow in size is attributable to two factors: the desire to provide more detailed information on subareas and the continuing efforts to better explain behavior of variables through new theories or by adding new data and variables. The linking of local systems reflects new interactions that have not been considered in the local systems. It is important that the linking be performed automatically, as manual linking introduces errors, which may be confused with the interactions between local systems.

3. The MODEL Language and Processor.

3.1. Specification Language.

The basic MODEL language essentially includes two types of statements: equations and declarations describing an organization of databases and/or reports. The MODEL compiler produces an efficient program in PL/1 for performing the specified computations. The MODEL language has several advantages for use in specifying local and global computational systems. It is non-procedural in the sense that the statements are unordered and entirely descriptive (no commands). No iterations, memory allocations or input/output need to be specified. The user need not be concerned with the efficiency of the computation, which to a large extent is determined by the automatic program generator. Thus the requirement of expertise in programming is replaced by a need for good mathematical background, which is common in the Sciences and Engineering. The numerical and statistical methods employed may optionally be suggested by the user, otherwise the need for them is recognized by the program generator and they are automatically incorpprated in the generated program. There is a pronounced enhancement in the ease and naturalness of expressing a computation in this manner as compared to conventional programming languages. For example, we have experienced a ratio of 5:1 of the number of resulting object program lines in PL/1 to the number of source lines in MODEL. This is indicative of the amount of detail which a system spares the user. One of the objectives of the research has been to increase this ratio.

The medium of equations is natural to scientists and engineers. For example, equations are the general mode of expression in economics. It is therefore common in engineering to use schematic diagrams of systems, circuits or networks as the medium for input to Computer-Aided Design (CAD) systems. However, these schematic diagrams are generally translated by the CAD system into equations. Composing equations is, however, a laborious and error-prone task. It is also necessary to provide a variety of higher-level modes of expression in which a single equation using higher-order operations may replace a group of equations using lower-order operations. We have explored the use of algebras where operations are applied to entire data structures (arrays and trees). In this way a single equation using high-level operation may replace a number of equations with elementary operations. The high-level operations of matrix algebra are a natural candidate for inclusion in the specification language. The operations in this case are applied to entire matrices of vectors. Matrix algebra is particularly powerful in defining regression analysis methods which are widely used in the envisaged applications. We also found that many of the encountered computations contain manipulation of databases. We have therefore designed operations on entire files, such as SELECT, MERGE, or SORT. Another avenue is to add relational algebraic operations such as PROJECT and JOIN.

3.2. Internal Representation of a Specification.

It is necessary to represent a specification in a convenient form, based on which implicit information is derived, checks are conducted, and finally a schedule of program events and a corresponding program in PL/1 are generated. We have followed the conventional approach to this class of problems by using a form of a directed graph to represent the dependencies and other relations involved in the computations. However, the straightforward approach of constructing a graph in which each element of an array variable or each instance of computation of an equation is represented by a separate node is unacceptable. First, the number of elements in an array may vary and not be known at the time of program generation. Second, the array may be so large as to result in a huge unmanageable graph. Consequently, we have developed a new tool which we called an array graph, where the nodes and edges represent entire arrays of data or equations and their relations. To retain the information on each element of an array node, we associate with each node and edge in the array graph additional information consisting of dimensionality, ranges (sizes) of dimensions and the forms of subscript expressions used. The array graph, together with the associated information support the deep analysis necessary to check the specification, recognize the need for special solution algorithms and derive an efficient schedule for a program to perform the respective computation.

3.3. Checking and User Interactions.

A specification can be checked on a number of levels, starting from checking individual statements to checking consistency and completeness of the entire specification. Syntactic and semantic checks are first performed on individual statements. Some corrections may be made automatically. In more difficult cases, the user is requested to make appropriate corrections.

Of much greater interest are the specification-wide checks of completeness, non-ambiguity and consistency. The process of checking the entire specification may be essentially regarded as inferring or "propagation" essential information from node to node. Missing statements may be generated or statements with missing parts may be completed. The user is notified where corrections have been made automatically or where user intervention is necessary. The problems discovered are described to the user in non-procedural terms, i.e., without referring to programming details. The following general types of analysis take place:

1. Dimensionality, subscripting, size of dimensions and data types of variables are checked through propagating them from node to node.

2. Data description statements are generated for variables referenced in the equations but not described by the user.

3. Equations are generated to relate same named input and output variables.

4. Solution methods for simultaneous equations are indicated wherever a set of simultaneous equations is detected. Similarly, optimization methods are included when there are more unknowns than equations, and an objective function and a set of constraints are defined by the user.

Our experience to date has indicated that these checks are very powerful in locating user errors. This aspect of the system is very important to achieve reliability of the computations, as large-scale systems are particularly prone to the occurrence of errors.

3.4. Recognition of Need for Statistical and Numerical Methods.

Two problems areas are forseen here. First, a user composed specification may not be complete because it does not explicitly specify some needed algorithms. We are not proposing to develop new computational algorithms, but instead to incorporate in the library of the system the most widely used algorithms in the areas of solution of simultaneous equations, optimization and regression analysis. With each of these algorithms would be provided a method of analysis of the array graph which would recognize the need for use of the respective algorithms. The user would also be able to suggest that selected algorithms be employed. In this case the system will check that these algorithms are indeed appropriate.

The system generates a custom-tailored subprogram to implement the selected algorithm and incorporates it in the overall program. There are two approaches to the implementation. One approach is to define the algorithm in the source MODEL language. In some cases this approach leads to a more efficient program as the operation of the algorithm can be better integrated with the rest of the specification. However this results in a larger specification. The other approach is the insert the algorithm in one piece during the PL/1 code generation phase. Both methods have been utilized, depending on the specific situations.

The second problem area concerns obtaining a solution which would be satisfactory to the user. Because of considerations of efficiency and also because the equations may be non-linear, it is generally necessary to employ iterative methods to perform solutions or optimization computations. Iterative methods of solution may not converge or may lead to unsatisfactory solutions. In such cases, it is necessary to provide information to the user on the equations and variables that are involved, and on the progress of the solution. The user must be able to recognize, based on this information, the source of the problem and make appropriate adjustment in the equations, the method used, or in the initial conditions. These changes must be expressed in the specification of the computations.

3.5. Efficiency of Produced Programs.

The capability for producing efficient programs is a key requirement because of the large-scale of the computations. Research in program efficiency has generally been very difficult and laborious, due to the complex analysis and modelling that is necessary for evaluation of efficiency approaches. Several aspects of this problem and respective approaches are briefly reviewed below.

First, we have been implementing the intelligent system for distributed cooperative computation using the PL/1 language rather than other languages which might be easier to program but would be inefficient in execution. Despite that, it would be a minor undertaking to produce the programs in another compiler language, such as FORTRAN. The overall system is very large (presently approximately 20,000 statements in PL/1) and consumes significant amounts of computer time to generate a program. For example, a specification of approximately 200 MODEL statements, from which a program of approximately 1000 PL/1 statements is produced, requires two to four times more computer time than the PL/1 compilation of the respective program. This is still within an acceptable range of computing costs, taking into account the present tradeoff of costs of computer time and software. Much larger specifications with thousands of equations require much more computing time, and they must be partitioned into smaller local systems.

In composing a specification of a computational system, the user chooses a natural and easy representation. Typically this choice does not correspond to the most efficient method of computation. It is up to the system to map the user's representation into an efficient procedural computer program. An overall flow of program events is produced in a skeletal, object language independent form called a schedule. The final program generation phase translates individual entries in the schedule into statements in the object language. Thus, the user ordering of the statements in the specification is not significant. Also the user description of data is independent of the medium of the data. i.e., whether it is in internal (core) or external (secondary stage) memory, or in the form of messages received through communication lines. The system determines the schedule based on the array graph and the information associated with respective nodes and edges.

In the case of an equation defining elements of an array variable, the schedule calls for repetitive calculations of the equation for all values within the ranges of the subscripts. Thus the schedule encloses equations within repetitive loops which might be nested if the respective arrays are of multiple dimensions. Determining the scope of iterations is a key factor in achieving efficiency in use of memory and computation time. Generally, maximizing the scope of iterations enables sharing memory locations by elements of the same or related array variables. However, in some complex cases this rule is not sufficient and it is necessary to actually compute the memory requirements of different candidate scopes of iterations to select the one which is most economical in use of memory.

Maximally-strongly-connected components in the array graph imply the existence of one of several cases as follows: simultaneous equations, optimization, or that the solution sequence cannot proceed in the order of ascending subscript values. An iterative method may be applicable in these cases. A user may optionally designate a set of simultaneous equations as a block (possibly nested blocks) and provide initial values, convergence conditions, and maximum number of iterations in the numerical solution. In the absence of such specification, the system recognizes automatically the existence of simultaneous equations and selects default values for the parameters of the iterative solution. Straightforward iterative solution of all the equations may result in a very large sparse set of simultaneous equations which is very inefficient to solve. It is necessary therefore to find dense clusters (or have the user specify blocks) of equations in each local computational system. Then each cluster or block is solved iteratively, repeating the solutions of all the clusters or blocks until the changes in values of the variables are within the convergence limits. The iterative solution of a block or a cluster of simultaneous equations is nested within local or global system iterative solutions. The iterative solutions of all the local systems may be conducted in parallel, using the computers of the developers of the respective local systems.

3.6. Scheduling Local and Global Systems.

The generation of a schedule for each local system was briefly reviewed in the previous section. Computation of each local system is performed sequentially following the order of events specified in the schedule. However, the local systems in a global distributed configuration are computed in parallel in respective computers, whenever possible. Local systems are viewed as data driven. i.e., they each read and await receipt of data from other local systems and then proceed with computation and send data to other local systems. Of major concern is assuring that deadlocks do not develop where local systems circularly await for input on which their output depends. One possible approach is to provide a monitor program that will synchronize all the local systems on a dynamic basis at runtime and recognize development of deadlocks. This, however, results in inefficiencies and does not provide the user with sufficient information to explain the reasons for deadlocks. We have elected to schedule and synchronize all the local systems a priori, to prevent the occurrence of a deadlock.

The global schedule is generated using the same methodology as for the local schedule. There is no need to have the same level of detail in generating a global schedule as for the local one. In both cases the input is essentially the respective array graph. In the case of a local system, the input is the array graph of the respective specifications. In the case of the global schedule, the input is a reduced array graph of the entire global system, where a node, or few nodes, represent an entire local system.

It is necessary to coordinate the local schedules and the global schedules to reflect the same dependencies and synchronization. For this reason, we generate with each one of the local schedules also a set of precedence constraints that reflect the order of the communications of the local system with external local systems. These precedence constraints are used optionally as an additional input in generating the global schedule. In this way we assure correspondence between any two schedules. The process can also proceed in the other direction, generating first a global schedule and obtaining precedence constraints which are then used as an additional input in generating each one of the local schedules. This is further discussed below. It can be shown that a deadlock-free synchronization can be obtained in this way, if the local systems are well-defined. This involves a fairly complex theory and it would be necessary to investigate and verify it by test of the interactions between local systems in the context of a large-scale computational system.

3.7. Applications.

The MODEL system has been used first in various accounting systems to investigate applicability to business data processing. More recently the system has been used in large-scale econometric modelling to investigate applicability to sciencific computations.

The business data processing use stressed large-scale systems where the atuomatic program generation is used to best advantage. The initial external use of MODEL was by the Internal Revenue Service in converting data on payments from diverse companies into a variety of standard statistics and formats. Programs were generated based on diverse data organizations employed by approximately 50 companies.

An on-going use of MODEL involves conversion, updating and redesign of approximately 500 COBOL programs by the Navy Accounting and Disbursing Center in San Diego, California.

There were also several experimental uses of MODEL that included development of complex accounting systems to investigate the ease in changing accounting rules to respond to new requirements.

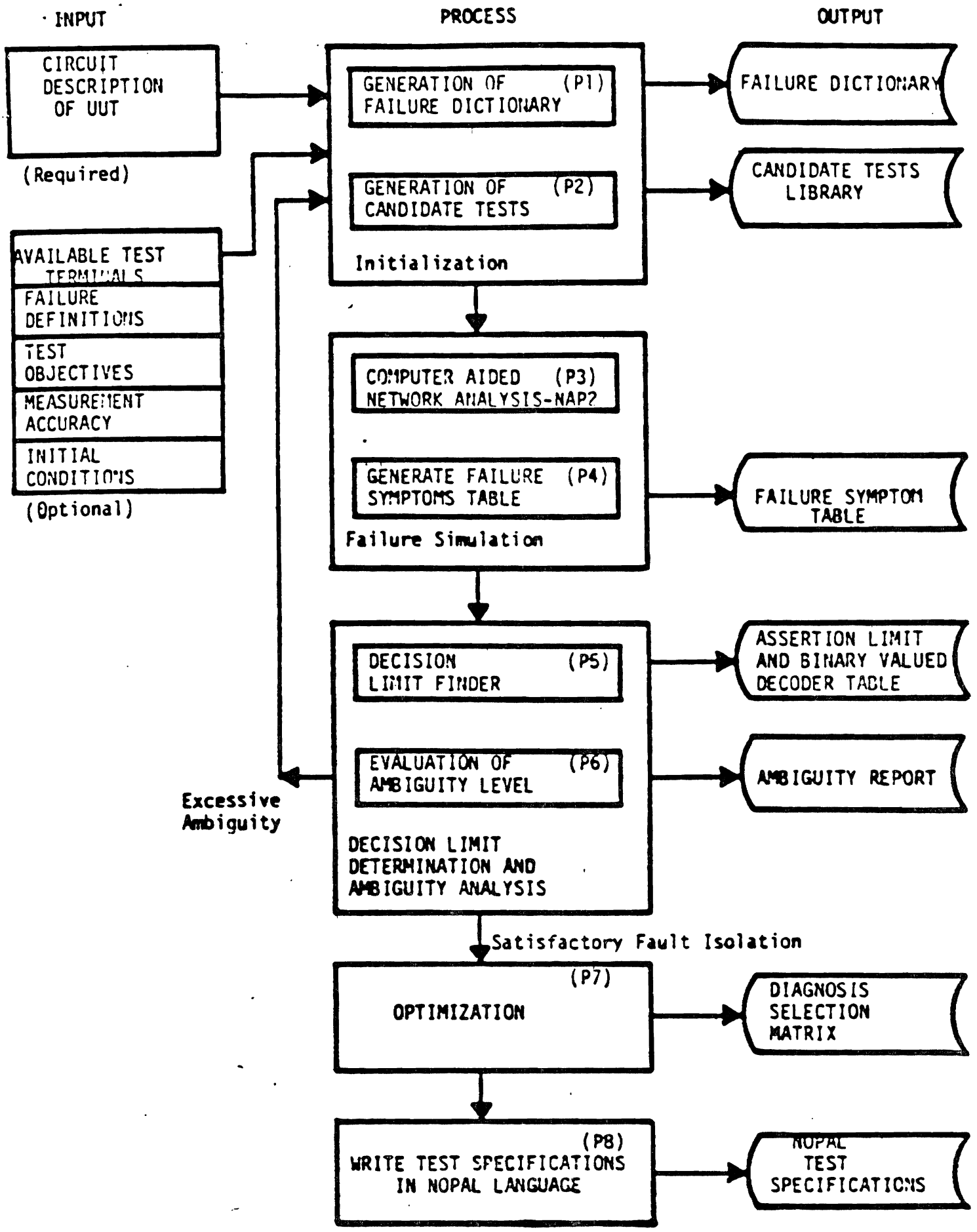
The use of MODEL in econometric modelling has been conducted in connection with Project LINK, which is a cooperative project involving some 25 reseach institutions that jointly develop a world-wide econometric model. This application stressed large-scale computations, involving tens of thousands of equations and diverse data bases. It also involved linking of independently developed country or region econometric models to form a world-wide model.

4. Description Of The NOPAL System.

Automatic test systems (ATS) may be viewed as consisting essentially of three main components: 1) The automatic test equipment (ATE), 2) The software and 3) The unit under test (UUT). We will focus on the software for ATS, which consists of test programs for each individual UUT that is to be maintained. A large number of programs are needed for the great diversity of UUTs. The cost of ATS software greatly exceeds the costs of the ATE equipment. The first task in developing a diagnostic test program for a UUT consists of analysis of the UUT, determination of the effect of failures of components and design of tests by which these failures may be isolated and diagnosised. It is then necessary to prepare an efficient test program that will be used with the ATE to perform the tests and produce appropriate diagnoses. These two tasks are enormously complex and laborious. It has been necessary, therefore, to develop computerized aids for their performance. While there has been extensive activity in developing computerized aids for testing digital circuits, there has been relatively little development of computerized aids for testing analog circuits. The NOPAL system is addressed primarily to ATPG for analog or hybrid (analog/digital) circuits. We have developed a two part NOPAL system to perform the functions of the electrical engineer and programmer, respectively. The two systems are denoted as the Top Part and Bottom Part.

4.1 The Top Part.

The objective of the Top-Part of the system is to find a small and effective set of diagnostic tests for a UUT, and express it in the NOPAL language. The procss is illustrated in the figure below. The inputs required of the user are shown on the left side, the methodology and processes are shown at the center and the output reports are shown on the right.


INPUT
PROCESS
OUTPUT
CIRCUIT DESCRIPTION OF UUT
(Required)
AVAILABLE TEST TERMINALS
FAILURE DEFINITIONS
TEST OBJECTIVES
MEASUREMENT ACCURACY
INITIAL CONDITIONS
(Optional)
GENERATION OF FAILURE DICTIONARY (P1)
GENERATION OF CANDIDATE TESTS (P2)
Initialization
FAILURE DICTIONARY
CANDIDATE TESTS LIBRARY
COMPUTER AIDED NETWORK ANALYSIS-NAP2 (P3)
GENERATE FAILURE SYMPTOMS TABLE (P4)
Failure Simulation
FAILURE SYMPTOM TABLE
DECISION LIMIT FINDER (P5)
EVALUATION OF AMBIGUITY LEVEL (P6)
Excessive Ambiguity
DECISION LIMIT DETERMINATION AND AMBIGUITY ANALYSIS
ASSERTION LIMIT AND BINARY VALUED DECODER TABLE
AMBIGUITY REPORT
Satisfactory Fault Isolation
OPTIMIZATION (P7)
DIAGNOSIS SELECTION MATRIX
WRITE TEST SPECIFICATIONS IN NOPAL LANGUAGE (P8)
NOPAL TEST SPECIFICATIONS

4.1.1 Inputs.

There are six inputs: (1) circuit description, (2) accessible test terminals, (3) UUT failure definitions, (4) fault isolation testing objectives, (5) measurement accuracy, and (6) initial conditions. The first input is required and the remaining are optional.

Circuit Description of UUT: The analysis of the circuit is based on simulation of faults. The modeling is based on the schematic circuit drawing of the UUT. The schematic diagram may consist of resistors, capacitors, inductors, mutual-inductances, voltage and current sources, bipolar and FET devices and integrated circuits. Accurate modeling of complex components may be an involved task, however sufficient published data is available for popular components. Each circuit component is given a unique name, where the first letter identifies the component type. The value of any element may be defined by a numerical constant, table, or mathematical expression. Component tolerances are specified by stating the maximum percentage deviation from the nominal value. Each circuit node is assigned a name. Current flow direction and source polarities are also indicated. The status of mechanical switches or potentiometers are treated as different initial conditions of the system. The description of an equivalent circuit follows conventions used in Computer Aided Circuit Analysis (CANA) programs. We use the NAP 2 CANA and follow its conventions.

Availability of Test Terminals: Any of the circuit nodes may be used for attaching test devices. However, the user may restrict the class of terminals available for testing to external contacts on the circuit board.

Failure Definition: The objective of testing is to discover the components of the UUT that have failed in a manner defined by the user. Since failure is a relative concept, any deviation from a components' nominal value may be declared by the user to be a failure. Typically catastrophic (open and short circuit) are most common. To ease the tasks of input preparation, individual catastrophic failures are included automatically and the user has to declare only the remaining failure definitions (i.e multiple component failures) as changes to the nominal circuit description. The number of tests that are required is related to the number of failures. Testing to diagnose a large number of potential faults may be extremely time consuming and expensive. The user can compromise between cost and quality by restricting the test objectives to discover only the more likely or most harmful failures.

Fault Isolation Test Objectives: To reduce the number of tests and lower testing costs, the user may wish to accept tests which will sometimes not locate a failure in a specific component, but in a small group of components. The failure isolation requirement is expressed in statements denoting that Pk% of the total number of possible failures may be located in ambiguous classes consisting of k or less components. P's are cumulative percentages, therefore Pk1 < Pk2---Pkn and k1 < k2---kn.

Measurement Accuracy: Three types of accuracy may be specified (1) minimum measurement threshold, (2) percentage inaccuracy or the measurement, (3) number of significant digits of the measured value.

Initial Conditions: A final optional input section specifies also the initial conditions of the UUT to speed the computer solution.

4.1.2 Process.

The first component of the process (P1) creates a failure dictionary data base for the UUT, based on the UUT circuit and failure description supplied in the input.

The next component (P2) generates candidate stimuli and measurements for tests using the three strategies described below, one at a time. First, small voltage stimuli (d.c. or a.c., depending on the type of components involved) are connected to connecting - points of the UUT, with the objective of measuring impedances at the connection points. This is referred to as the cold-circuit strategy. Next, the UUT is powered with the nominally specified d.c. power sources, and voltage and current measurements are conducted at the available nodes. This strategy is referred to as a d.c.-nominal. Finally, an a.c. signal is applied to input connecting points and user specified tests are conducted. This strategy is referred to as a.c.-signal. The system design process provides for addition of more test strategies in the future. These strategies are all employed one at a time in the above order. An enhancement of adding a strategy of applying multi frequency stimuli is part of this proposal, described in Section 4.

Next in (P3), the circuit behavior is simulated with the above stimulus applied, with the components having nominal values and with the components having failure conditions enumerated in the failure dictionary, one at a time. The sensitivity of the circuit response due to tolerances is also determined for each case. A CANA program, NAP2 has been selected to perform the simulation based on considerations of economy of computer usage costs. The simulation produces, for the nominal case and for each failure, ranges of measurable physical entities (voltage, current, phase, etc.) observed at connecting points.

In the 4th component (P4), each range to be verified by a measurement is specified in an assertion. These assertions, together wtih information on the associated connecting points for the stimulus and measurement and on the associated failures are inserted into a Failure Symptom Table.

Based on this information, it is possible in P5 and P6 to verify if the tests formulated so far meet the ambiguity requirement statements set forth in the input. If testing objectives are still not met, the next strategy is employed, new tests are examined and the circuit is simulated until the above criterion is met, or until all test strategies have been exhausted.

When it is determined that the ambiguity levels of fault isolation are satisfied, the P7 component is initiated. The purpose of process P7 is to reduce the number of tests to be performed without loss of fault diagnosis and isolation capability. There are three optimization steps. First, the total number of test setups is minimized since setup for a test consumes the longest time in actual testing. Second, the available assertions of the remaining tests are further reduced. Finally, diagnosis logic is found such that only the minimum number of tests in the remaining set are performed to diagnose and isolate each failure or group of equivalent failures.

4.1.3 Outputs.

The reports produced by the Top-Part give a step by step picture of the progress of circuit analysis and test design. This reporting gives the user sufficient information in order to overcome a variety of problems that may arise.

Failure Dictionary: The failure dictionary consists of the catastrophic failures of the individual components of the circuit and any other failure modes specified by the user.

Circuit Analysis Output: This is a data base consisting of the unmodified outputs of the NAP2 system for each simulation of the circuit.

Stimulus, Measurement-region and Failure Tables: This is a series of tables which show the tests initially selected and progressively those retained or consolidated in the test optimization process, as well as the basis for delection of other tests.

Ambiguity Report: An elementary group of failures where it is not possible to distinguish further between the individual (or subgroups of) failures is referred to as an equivalent class of failures. This report is in a form of a table with a row for each equivalent class. The columns consist of an identification number, the number of failures (referred to as the ambiguity), the percentage of the total number of failures, a cumulative percentage, and a list of the component names and their failure functions constituting the equivalence class.

NOPAL Test Specification: This report consitiutes the final documentation of the test requirements. The NOPAL test specification is shown and explained in the next section. In addition there are several summary reports.

4.2 The Bottom Part.

The Bottom Part can be used by itself, where the user composes the NOPAL language specification, or can be used in conjunction with the Top Part, where the NOPAL specification is generated automatically. In the former case, use of NOPAL is advantageous over programming directly in ATLAS.

The major part of a NOPAL specification consists of specifications of individual tests. Some description of the tests and diagnoses for the UUT must be available prior to the user composition of a NOPAL specification. Test descriptions are typically available in technical or maintenance manuals for the UUT or in specifications of acceptance tests. If a good description of these tests exists, the programmer needs only to restate them individually in the NOPAL language. The derivation of tests frequently requires deeper analysis of the UUT. Such analysis can be performed manually or by the Top Part of NOPAL.

If the description of the tests is not available from the Top Part, then the programmer specifies these tests in NOPAL, one at a time, in an arbitrary order. There is no need to consider the order of execution of tests. Each test is relatively independent of the others. Further, a test is divided into relatively independent subparts. Each test contains six subparts: stimuli to be applied to the UUT, measurements to be taken, relations to be evaluated, logic for selecting the diagnoses, and the respective interaction with the operator through messages and responses. The programmer need not be concerned about the ordering of the tests or the ordering of statements within a test. There is also no need to declare internal variables or provide input/output and iteration statements. The selection of algorithms that are efficient in use of computer time and memory, normally performed by an ATLAS programmer, is performed automatically by the NOPAL processor.

The user of NOPAL need not be concerned with selection of ATE devices and connection points. Thus, it is not necessary that a user possesses this type of expertise. Instead, the user refers to functions defined elsewhere without requiring knowledge of the stimuli, measurements or computations that are involved.

Global checking of the specification: in addition to the analysis of the syntax and semantics of individual statements, the NOPAL processor performs global analysis to detect incompletenesses, ambiguities or inconsistencies in the overall specification. These checks assure that all the variables have been defined, that ambiguous naming of variables has not occurred, and that there is consistency in dimensionality of variables and in the use of subscripts of array variables. The Bottom Part of the NOPAL system attempts in some cases to correct the specification automatically, while producing warning messages for the user. In other cases, error messages are issued, explaining the problems discovered and soliciting corrections from the user. Since the NOPAL language is non-procedural it is much easier to perform these checks on a NOPAL specification than on a program in a procedural language such as ATLAS. The global checking in NOPAL should lead to a more reliable and complete program.

Debugging and Modification: once a specification has been completed and an ATLAS program generated, the user can run the produced program on the ATE with a UUT to check whether the program performs according to the intentions of the programmer. Initially the user may generate an ATLAS program that includes a trace of all activities. The

ATLAS program with a trace includes the output of results of all measurements and tests. Another version, without a trace is generated for productive use Deviations of output variables from what the user expected may be observed by running the program. Due to the non-procedural nature of NOPAL, the order of the statements is immaterial, and a variable can have only a single value. There is no need in the debugging to find the appropriate place for the modifications or to trace the different values that are stored in a variable memory location during the execution of a program, as is the case when debugging or modifying an ATLAS program. The user must only examine the statements that define the respective variables. If an error or a change is desired, it is necessary to modify these NOPAL statements and generate a new program in ATLAS.

Documentation: the Bottom-Part produces three types of documentation which are used in debugging and modifying a specification. The first type consists of a listing of the NOPAL specification, which has been reformated for easy readability. The individual parts of the specification are clearly identified and an indentation scheme is used to indicate the relation between parts and constituent subparts. Next, there are a variety of cross-reference reports. One cross-reference report shows the usage of all identifiers (i.e. variables, connections, stimuli and measurement functions, etc). Other reports cross-reference various components of the specifications, such as the diagnoses selected by respective tests, or the failures of the UUT detected by respective tests. The list of failures allow the user to check that the program indeed tests for all the types of failures which the program is intended to diagnose. The final type of report consists of flowcharts for each of the tests and for the overall program. This type of report is sometimes helpful in debugging, as it also shows the various dependencies between variables and statements.

Maintenance: the documentation generated when a NOPAL specification is compiled serves as a basis for maintenance. The formatted NOPAL specification provides a maintenance programmer with a concise description of the tests. The specification may be modified to meet the changing needs and submitted to the NOPAL processor. The modification may involve modifying, deleting or adding tests. A new ATLAS program and accompanying documentation are then produced. Thus, there is no need to refer to the ATLAS program either in the original development of the program or in the subsequent maintenance. The documentation produced with each updating of a test program is then retained to serve as a basis for future maintenance.

References

1. "Automatic Test Design", Cihan Tinaztepe, Technical Report, Submitted to U.S. Army Electronics Command, June 1978, Ph.D. Dissertation in Computer Science, University of Pennsylvania, 1977.

2. "Verification and Correction of Non-Procedural Specification in Automatic Generation of Programs", Technical Report, Submitted to ONR, September 1978, Ph.D. Dissertation in Computer Science, University of Pennsylvania, 1978.

3. "An Automatic Program Generator for Model Building in Social and Engineering Science", Technical Report, Submitted to ONR, October 1978 by Jorge L. Gana, Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, 1978.

4. "Generation of Software for Computer Controlled Test Equipment for Testing Analog Circuits", C. Tinaztepe and N.S. Prywes, IEEE Transactions on Circuits and Systems, Special Issue on Automatic Analog Fault Diagnosis, June 1979.

5. "Use of a Non-Procedural Specification Language and Associated Program Generator in Software Development", N.S. Prywes, A. Pnueli and S. Shastry, published in TOPLAS, October 1979.

6. "Use of NOPAL in Generating Programs for Testing Analog Systems and Circuit Boards", C. Tinaztepe and N. Prywes, Proc. of the IEEE International Conference on Circuits and Systems, October 1980, pp. 924-927.

7. "Design Report on Operations on Arrays and Data Structures, Amir Pnueli and N.S. Prywes, Prepared with Support from the National Science Foundation, Joint Computer Science-Economics Project, Department of Computer and Information Science, Moore Scool, December 1980.

8. "Modularity in Non-Procedural Languages Through Abstract Data Type", Rajeev Sangal, Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, August 1980.

9. "Compilation of Non-Procedural Specifications into Computer Programs", N.S. Prywes and A. Pnueli, Prepared for Informations Systems Program, Office of Naval Research, (Automatic Program Generation Project), Moore School, April 1981.

10. "Distributed Processing in the MODEL System--with an Application to Econometric Modelling", by Amir Pnueli and Noah Prywes, Prepared with support from the National Science Foundation under Grant MSC-79-0298 (Joint Computer Science-Economics Project), Moore School, June 1981.

11. "Programmer's Guide to the Top Part of NOPAL", by Cihan Tinaztepe, Prepared under Contract DAAB07-80-F1690 for U.S. Army Communications and Electronics Material Readiness Command, Moore School, August 1981.

12. "Simultaneous Equations in the MODEL System with an Application to Econometric Modelling", by Richard Gary Greenberg, Prepared with Support from the National Science Foundation, under Grant MCS-79-0298 (Joint Computer Science-Economics Project), Moore School, October 1981.

13. "Program Optimization Based on a Non-Procedural Specification", by Kang-Sen Lu, Ph.D. Dissertation in Computer and Information Science, University of Pennsylvania, Prepared under contract N00014-76-C-0416, from Information Systems Program, Office of Naval Reseach, Moore School, December 1981.

14. "MODEL Program Generator: System and Programming Documentation", by Kang-Sen Lu, Prepared under Contract N00014-76-0-0416, from Information Systems Program, Office of Naval Research, Moore School, May 1982.

15. "NOPAL Reference Manual: Bottom Part", by Noah S. Prywes, Prepared with support from U.S. Army Communications and Electronics Command, under Contract DAAB07-81-F-1598, Moore School, May 1982.

16. "Automatic Program Generation in Distributed Cooperative Computation", by Noah S. Prywes and Amir Pnueli, June 1982.

17. "Compilation of Non-Procedural Specifications into Computer Programs", by Noah S. Prywes and Amir Pnueli, July 1982.

18. "Checking of Program Specifications in the MODEL System", by Evan Lock and Noah S. Prywes and Amir Pnueli, July 1982.

19. "Automatic Verification and Debugging of a Program Specification", by Boleslaw Szymaski, August 1982.