AD-A095 546 MOORE SCHOOL OF ELECTRICAL ENGINEERING PHILADELPHIA P--ETC F/G 9/2
MODULARITY IN NON-PROCEDURAL LANGUAGES THROUGH ABSTRACT DATA TY--ETC(U)
AUG 80 R SANGAL N00014-76-C-0416

UNCLASSIFIED NL

1 OF 2
AD A
095546

LEVEL II

2

AD A095546

DTIC
ELECTE
FEB 26 1981

UNIVERSITY of PENNSYLVANIA

The Moore School of Electrical Engineering

PHILADELPHIA, PENNSYLVANIA 19104

DDC FILE COPY

81 1 16 044

UNIVERSITY of PENNSYLVANIA

2

PHILADELPHIA 19104

The Moore School of Electrical Engineering D2

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE

MODULARITY IN NON-PROCEDURAL LANGUAGES THROUGH ABSTRACT DATA TYPES

By
Rajeev Sangal

Prepared with Support:

Under Contract N00014-76-C-0416
Information Systems Program
Office Of Naval Research
Arlington, VA

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A095 546	3. RECIPIENT'S CATALOG NUMBER 9
4. TITLE (and Subtitle) MODULARITY IN NON-PROCEDURAL LANGUAGES THROUGH ABSTRACT DATA TYPES		5. TYPE OF REPORT & PERIOD COVERED Final Report
		6. PERFORMING ORG. REPORT NUMBER Moore School
7. AUTHOR(s) RAJEEV SANGAL		8. CONTRACT OR GRANT NUMBER(s) N00014-76-C-0416
9. PERFORMING ORGANIZATION NAME AND ADDRESS UNIVERSITY OF PENNSYLVANIA The Moore School of Electrical Engineering Dept of Computer Information Sciences		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Information Systems Programs Office of Naval Research Arlington, VA		12. REPORT DATE 31 AUGUST 1980
		13. NUMBER OF PAGES 154
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS.
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

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
programmer productivity
modularity
Nopal

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

This dissertation presents abstract data types as a means of introducing modularity in non-procedural languages. Non-procedural languages based on equational specifications have been proposed in recent years to improve programmer productivity reliability. Issues of structured programming (i.e. disciplined use of the control structure) have no meaning in the context of

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

KEYWORDS cont.

Nopal processor
Equate-Atlas
Equational specifications

ABSTRACT cont.

these languages because these are devoid of any control structure. Statements in a specification can be given in any order; the sequence of execution is determined after an analysis of the specification. Modularity, however, still remains an important issue in the context of these languages, as it allows specifications to be written and processed independently. Abstract data types are proposed as a means of introducing modularity. Notion of module for the specification of abstract data types is introduced and its denotational semantics is given. Nopal, a non-procedural language for the specification of testing of electrical circuits, has been chosen in which abstract data types are introduced for modularity. The abstract data types also allow specification of virtual devices in testing. An implementation of the Nopal processor is given. The Nopal processor analyzes a Nopal specification for completeness, consistency, and non-ambiguity; and generates a sequential program in Equate-Atlas corresponding to the specification. The various phases of the Nopal processor for the analysis of a specification are described. Finally, some example specifications together with their generated Equate-Atlas programs are given.

MODULARITY IN NON-PROCEDURAL LANGUAGES THROUGH ABSTRACT DATA TYPES

Rajeev Sangal

A DISSERTATION

in

Computer and Information Science

Presented to the Graduate Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirement for the degree of Doctor of Philosophy.

31 August 1980

[illegible]

Supervisor of Dissertation

Graduate Group Chairperson

To Amma and Papa

Acknowledgment

I am thankful to my advisor Dr. Prywes for his guidance and encouragement during this research, to Dr. Amir Pnueli for his penetrating comments, to Dr. Peter Buneman for his encouragement especially during the early stages of the work, and to other members of my committee Drs. A.K. Joshi, H.L. Morgan, and T. Finin for reading the dissertation and raising interesting questions.

Her daw Che and Kang-Sen Lu, who were my officemates during the early and later stages of my work respectively, contributed in many ways. They were always ready to lend me their ears. Thanks are also due to Dr. C. Tinaztepe for his help during the early stages of my work.

Study in America has been a rich and rewarding experience for me. I have learnt a great deal both in academics and otherwise. It has given me a better understanding of India, my country, and its culture; and of my own self. Towards the promotion of this understanding, I wish to thank Balaram, Jasenka, John, Mladen, Neeraj, Nikhil, Poonam, Pradeep, Pradip, Prasad Rao, Raj, Ramarao, Samar-da, Swapan, and Venky.

Table of Contents

Table of Tables

Table of Figures

Chapter One

INTRODUCTION

1.1 MOTIVATION

The software crisis of the sixties saw the acceptance of structured programming, modularity, and top down design methodologies in building and maintaining software systems. The underlying philosophy behind these methodologies was that the software systems are complex; that they are hard to understand and difficult to manage; and to keep them within manageable limits, the discipline of structured programming should be imposed on the programmers. It reflected and still reflects the state of software technology. The requirements for systems are specified informally or semi-formally to the programming team, which then implements a system satisfying the requirements. There is a large gap between the specification language (generally, English for informal specifications) and the implementation language (Fortran, Cobol etc.), thus causing the implementation to address a lot of detail. This leads to increased complexity of software which makes the debugging and maintenence difficult. The structured programming and top down approach accepts this complexity as unavoidable, and tries to keep it under control by requiring the programmer to use simple program structures.

Continued growth in the size of the software systems, the demands of reliablility and programmer productivity requires new solutions. It has led to activity in the field of, what are called, *very high level languages* (VHLL). These languages reduce the gap between the informal specifications and programs. Sometimes, these languages are of sufficiently high level that the program itself is the specification satisfying the (intuitive or mental) requirements. Some of these languages, in which order of statements in programs is immaterial, are called *non procedural languages*. In fact, a program in these languages is so unlike a program in procedural languages that we call it a *specification*.[1] Many of the issues of structured programming (e.g. disciplined used of the control structure) no longer have any meaning in the context of non-procedural languages, since the specifications do not have any control structure. The details relating to the control are no longer the concern of the user, but rather, are handled by the compiler for the language.

Modularity still remains a useful and important issue for large specifications. *Modularity* may be defined as independence in compiling and composing of different parts of a larger specification. It is desirable because it simplifies the specification. This simplification results not only because of reduced size, but also because, with proper sub division, the smaller specifications represent logical sub-units of the larger specification. Modularity allows incremental development of a large specification. It also lends itself to easier modification. In most cases, only a few of the smaller specifications need to be changed when the needs of the specified system evolve or change.

1.2 BACKGROUND: MODEL AND NOPAL SYSTEMS

Model and Nopal are non-procedural languages developed at the University of Pennsylvania in an attempt towards a simple yet powerful very high level language. These languages have no control structure, and are based on the familiar notions of mathematics.

[1]"Specification" has also been used in the literature to express the "requirements" of a system, or the set of algebraic axioms defining an abstract data type etc. Usage of the term here should not be confused with its other meanings.

The Model system [42] has been designed to automate the generation of software for data processing applications. The first step is to provide a data processing requirements. It consists of three main parts. The first part is the header, which consists of the name of the specification and names of the data bases. The second part, the data description, consists of descriptions of the structure of the source and target data in the specification. The source data corresponds to the input data, generally on sequential and indexed sequential files; the target data refers to the desired output files. The third part, a set of assertions, specifies the relations between the source and target variables. There are no control statements typical of procedural high level languages, e.g. those that deal with input/output, loop control etc.

The Model processor analyzes many aspects of the specification. It checks for ambiguities, incompleteness and inconsistencies and issues appropriate messages to the user. It also generates a number of reports which serve as the documentation for the specification. The processor then produces a sequence of execution for the assertions, with appropriate loop control statements. Finally, it produces a PL/I (or Cobol) program.

The Nopal system [46] has been designed to automate the generation of programs for automatic testing of electronic circuits. A specification in Nopal has three major parts. The first part gives the test specification, the second part the unit under test (UUT) specification, and the third part the automatic test equipment (ATE) specification. These parts can occur in any order in the specification. The test specification consists of a number of tests each of which is used to specify the stimuli to be applied, measurements to be made, computations to be performed, and diagnosis to be selected. The specification of the individual tests is non-procedural, and similarly, there is no sequence specified between the tests. The diagnoses are normally selected based on the outcome of tests. They are used to isolate faulty components and print appropriate message to that effect. The UUT and ATE specifications are used to specify the characteristics of UUT and ATE.

The Nopal processor, similar to the Model processor, analyzes the specification for ambiguity, incompleteness and inconsistency. It too generates reports which serve as the documentation for the specification. The processor produces a sequence of execution for the tests, in the phase called inter-test sequencing. It then analyzes each of the tests individually and generates a sequence for the assertions, conjunctions, and diagnoses in the test. Finally, when all the problems are resolved it generates a program in Equate Atlas.

The issue of modularity is an important one for both the systems. At present the specification must be submitted as one unit. It leads to many of the problems mentioned in the previous section, and to some very practical problems when the processors for the language run out of address space during execution.

1.3 CONTRIBUTIONS

This dissertation examines and proposes the approach of abstract data types for modularity in these languages, and describes the implementation for the Nopal language. The following are the contributions of this work:

It has led to:

1. the definition of a scheme for modularity in non-procedural languages,
2. a novel way to define the abstract data types, namely, by means of the non-procedural specification, and
3. automatic generation of program modules that correspond to respective specification of the abstract data types.

Abstract data types provide a non-procedural way to introduce modularity. Variables in the specification can be declared to be of abstract type, in which case they may be operated upon by a restricted set of functions. The definition of an abstract data type along with the set of functions is given separately by means of a "module". The specification of a module is given non-procedurally, leading to the dual contributions (1) and (2).

Finally, the above ideas on modularity are used in the Nopal system The Nopal language has been developed to generate programs for testing of electronic circuits. The abstract data type facility is used to define the devices for testing. My work on the above system has been on the development and completion of the original Nopal system [7], and implementation of the idea of abstract data types.

1.4 ORGANIZATION OF THE DISSERTATION

This dissertation is divided into six chapters. Introduction is given in this chapter, followed in Chapter 2 by a survey of past work in the fields of non-procedural languages and abstract data types.

Chapter 3 contains the use and specification of the abstract data types in a non-procedural languages independent of either Mod or Nopal. The use and specification of "modules" is described. Formal semantics of the modules is given and similarity of the module specification with algebraic axioms is shown.

In Chapter 4 the language Nopal incorporating the above ideas is described. Features of Nopal for specification of automatic testing of electronic circuits are presented. Implementation of Nopal is given in Chapter 5, and the various phases of the Nopal processor are described. Examples of Nopal specifications and the reports generated by the Nopal processor are given in the Appendix.

Conclusions and ideas for future work are suggested in Chapter 6.

Chapter Two

SURVEY OF RELATED LITERATURE

2.1 SURVEY OF NON-PROCEDURAL LANGUAGES

Looking back at the development of computers we find a hierarchy of computer programming languages. The assembly level languages form the lowest level and the higher level languages such as Fortran, PL/I, Algol etc. form the next higher level. Both classes of languages are characterized as (a) procedural, and (b) domain independent. They are procedural because the individual statements are prescriptive and a program in the language consists of a sequence of such statements. These languages may be used in widely varied application areas and hence are called domain independent.

The next higher level languages are referred to as very high level languages (VHLL) and they may be sub divided into two groups. The first group consists of languages which are domain dependent e.g. Business Definition Language (BDL) [21]; the second group consists of domain independent languages with facilities to describe higher level concepts which allow the omission of many details. Examples of this group are: SETL [29] which allows manipulation of sets and relations, APL [27] which has many convenient operators for matrices, LISP [57] which works on lists etc. In the second group there are many languages that are descriptive and are devoid of any control facilities. This class of languages is referred to as non-procedural, because a "program" in these languages does not give a prescriptive sequence to be followed, but rather defines variables and their values in a sequence free manner. A "program" in these languages is so unlike that in procedural languages that we call it by a different name: "specification", as mentioned earlier in Chapter 1.

The goal of the VHLLs is to allow the user to express his or her problem directly in these languages, thus leading to automatic programming systems which accept the specification and generate a program corresponding to it. [42]

> "The ultimate expectation for automatic programming may be visualized as a user (no longer a 'programmer') making a few simple statements, to which the automatic programming system responds by spewing out a program of several hundred statements, already correct and satisfying the user's intentions."

Non-procedural languages have been around for more than a decade ([6], [26], [47], [52], etc) and they continue to be of current interest ([1], [4], [25], [42] etc.).

One of the early attempts by Tesler [52] defined lists and operations on lists. An important operation was *PRECEDING*, which was used to express the relationship of the current item in the list to the preceding item in the same list or some other list. The language was restrictive because recurrence relations between items in lists could be specified using only PRECEDING. Some of the other early languages were interpreted and hence slow in time and inefficient in memory space.

More recently, LUCID has been designed as a formal system in which programs can be written and their proofs carried out. "The proofs are easy to follow and straight forward to produce because the statements in a LUCID program are simply axioms from which proof preceeds by conventional reasoning [1]." Variables and their history of values can be defined. The history is defined as a sequence of values using the primitives FIRST and NEXT. They essentially allow the specification of one level loops. To allow nested loops, a function called LATEST is introduced. However, it clutters up the program; consequently, BEGIN-END blocks to nest iterations are included in the language.

The limitations of the LUCID language are the absence of arrays or any compund structures and the use of NEXT to define relationships between sequences of values. The latter implies that the relationships be known in advance at program writing time, and cannot be computed at run time. For example, it is not possible to specify that the current element depends on the kth previous element, where k is computed at execution time.

Non-procedural languages Model [42] [48] [50] [39] and Nopal [7] [41] [56] [46] allow relationships between array variables to be defined explicitly by means of indices. This makes the languages richer than LUCID. At the same time, they are compiled rather than interpreted. A brief introduction to them has been given in Chapter 1.

A recent proposal by Kessels [30] is to mix procedural and non-procedural approaches. In his approach, "block" is the basic struc which indicates the scopes of names, as well as the mode (non-procedural or sequential). A "valued-block" has a set of values. Besides these, ther are multi-state blocks which retain information after the exit from the block. Many of these features serve to increase the complexity of the language, and make it difficult to learn and use.

A number of domain dependent systems have been proposed. Some of them are described below.

Business Definition Language [21] is a very high level domain dependent language. It is aimed at the problems of business data processing. It assumes a model of the processes involved in the manual methods used in businesses and tries to mimic those. There are three components: one for defining the business forms, one for describing the business organization, and one for writing calculations. Using a graphics screen the forms may be defined. They serve both as input and output, as well as internal representation of information. These documents can be routed to different parts of the organization or stored in files. Computations can be defined on the elements in the forms. Essentially, it is a tabular language with special constructs to represent the domain of business.

PSI system developed at Stanford [16] uses a model based appoach like BDL. However, it has provision for incorporation of an independent domain expert module. Information about objects and their relationships in the domain is included in the module, thus freeing the user from defining commonly used terminology. The modules may be changed depending on the domain.

PROTOSYSTEM I has been developed by the Automatic Program Generation Group at M.I.T. [45]. It consists of two parts: The top part consists of a man machine interface, a knowledge base on business management etc. *The bottom part obtains a data processing* specification from the top part, performs system design, and generates PL/I code. The specification language used is SSL, which is non procedural and resembles Model.

There are other examples of domain dependent systems, most notably, APS developed at University of Southern California at I.S.I. [3], SBA [62] etc.

2.2 SURVEY OF ABSTRACT DATA TYPE SPECIFICATION TECHNIQUES

Data abstraction has been identified as a widely useful program unit by recent work in programming methodology. It has also been identified to be a unit for which formal specifications can be written easily. It can serve as a basis for modularity. Consequently, work related to data abstraction or abstract data types is reviewed here.

There are two approaches for giving the formal specification of abstract data types. The specification can be given either by means of an abstract model, or implicitly via descriptions of operations on the data types [35]. In following the first approach, the behaviour is actually defined by giving an abstract implementation in terms of another data abstraction or mathematical discipline. In the second approach, the class of objects is determined inductively from the operations. Usually, it is the smallest set closed under the operations.

Liskov and Zilles [35] have further classified the approaches for specification of abstract data types into five categories. The classification is based on the method used for specification, e.g.

1. use of a fixed domain of formal objects, such as sets, graphs or arrays;
2. use of an appropriate known formal domain;
3. use of a state machine model;
4. use of an implicit definition in terms of axioms; and
5. use of an implicit definition in terms of algebraic relations;

to specify abstract data types. The first two categories use the first, i.e. abstract model approach, while the remaining use the second, i.e. implicit definition approach. Some examples belonging to each of the categories are given below.

In the first category, a fixed domain of formal objects is used to provide a high level implementation of the desired abstract data type. For example, V-graphs were used by Earley [11] to represent instances of data structures. Operations on the data structure are specified either by expressions written in terms of primitive V-graph operations, or by means of pictures of V-graph transformations.

An appropriate known formal domain can be chosen to give the high level representation for the abstraction. Generally, this is some established mathematical domain. Hoare has used this approach to specify sets and subsets of integers [24]. The advantage of this approach is that a body of knowledge is available about the formal domain; on the other hand, it may not be suitable for representing the abstraction.

Parnas [38] has developed a technique and notation for viewing the abstraction as states of an abstract state machine.

Use of axiomatic descriptions to specify the abstractions falls under the fourth category. The axioms define equivalence classes over the set of all expressions. If the set of axioms are well chosen, the equivalence classes are unique. The axiomatic specifications are minimal and widely applicable, however, they are deficient with respect to comprehensibility.

Recently, an algebraic specification technique based on the algebraic construction, known as "presentation", has emerged as a popular one. The algebraic axioms are easier to understand than the general axiomatic specifications, and they too are representation independent. An algebraic specification has two components: syntactic and semantic. The syntactic component gives the domains and ranges of the operations on the abstract data type. The semantic component consists of set of algebraic axioms in the form of equations, which relate the operations to each other. An implementation may also be given for the data types. "An implementation of an abstract data type is an assignment of meaning to the values and operations in terms of the values and operations of another data type or set of data types [18]." A correct implementation must satisfy the algebraic axioms. The data types used in the implementation are also specified by means of axioms; and their implementation may again be specified if they are abstract types. The proof of the correctness of the implementation requires showing that each of the algebraic axioms for the data type is satisfied when the implementations are substituted in the axioms. The definition of equality interpretation for the implementation is needed for the proof. A general principle used in the proof is that of data type induction. It means proving some invariant property of the data type, and involves establishing the base step and the induction step.

Goguen, Thatcher and Wagner have described an initial algebra approach to the specification, correctness, and implementation of the abstract data types [14]. They describe a first order language (or Σ-algebra) using sorts and signature over sorts. They then define a category C of Σ-algebras to consist of Σ-algebras together with all the Σ homomorphisms. Alg_{Σ} is defined as a [53]

> "universe of discourse where the process of axiomatizing on the data types is going on. In particular, the free algebra in Alg_{Σ} provides a language in which to write down the axioms, and their homomorphisms tell us how to interpret the *axioms*."

Given the above algebra, the concepts of presentation and initial algebra are introduced; it is proved that the initial algebras are isomorphic leading to the main result: "An abstract data type is the isomorphic class of an initial algebra in a category of Σ-algebras." It provides a rigorous mathematical basis for the specification techniques using axioms.

2.3 LANGUAGES SUPPORTING ABSTRACT DATA TYPES

The use of axiomatic specifications is still far from practicable. It involves fair degree of mathematical expertise to formulate the axioms and to check their consistency and completeness. Consequently, the practical languages which allow the definition of abstract data types are still based on the abstract model approach (which includes categories (1),(2), and (3)) described in the previous section.

CLU language was developed at M.I.T. [34] to support the use of abstractions in program construction. It supports three types of abstractions: procedural, control, and data. It has a mechanism called "cluster" to define the data abstraction. A cluster is used to define the representation of a data type and the set of operations which can be performed on it. The representation may be given using variables whose data types are again defined in other clusters. In such cases, references are associated with these variables, and the actual data is stored in the clusters. The only way to access or modify this data is by means of operations defined in the cluster for the appropriate abstract data type.

In the implementation, the variables which are defined to be of abstract data type actually store references to data, while the data and its representational details are given in the cluster. It does away with explicit manipulation of pointers, yet allows an efficient implementation. However, it causes a change in semantics of the traditional assignment statement. In the example:

```
B := NEW;
A := B;
MODIFY(B);
```

the variables A and B are of same abstract type which has operations NEW and MODIFY defined in a cluster. The first statement causes B to be defined, but since the data type is defined in a cluster, B simply stores a reference to the data. In the second statement the same reference is stored in A. The problem is caused by the third statement. Modification of the structure pointed to by B causes the modification of A, as a side effect.

The above suggests two alternatives: either the notion that A and B are of abstract data type should be abandoned and they should simply be declared to be of pointer type; or the semantics of the assignment statement be redefined. In CLU, the latter approach is chosen and the assignment statement is defined to mean "renaming". In the example, the second statement will be taken to mean that A is the name of the same object as denoted by B. In a language which is side-effect free the above problem, of course, does not arise.

Concurrent Pascal has been developed by Brinch-Hansen [5] for the writing of concurrent programs. It allows the definition of monitors. A monitor defines a type whose instances may be created. The data associated with an instance can be accessed using that particular instance of the monitor. This restriction disallows recursive definition of monitors. Moreover, operations defined in the monitor can operate on variables only in one particular instance of the monitor. These restrictions may be justified for concurrent programming, however, the language is not discussed any further here, due to these severe limitations.

Chapter Three

ABSTRACT DATA TYPES IN A NONPROCEDURAL LANGUAGE

3.1 INTRODUCTION

This chapter describes modularity in a simple non-procedural programming language based on mathematical equations, through the use of abstract data types. The presentation is independent of the Model or Nopal systems, referred to previously. The objective in this Chapter is to keep the language simple so as to convey the concepts without being encumbered by details.

Section 3.2 introduces the non-procedural language. Alternative aproaches to modularity are discussed in section 3.3. Use of abstract data types is described in section 3.4; and their specification using *modules* is given in sections 3.5, 3.6 and 3.7. Finally, the semantics of modules is given in Section 3.8, followed by a summary in Section 3.9.

In short, this chapter describes the design rationale of abstract data types for non-procedural languages based on mathematical equations.

3.2 A SIMPLE NON-PROCEDURAL LANGUAGE BASED ON EQUATIONS

A specification in a non-procedural language basically consists of two kinds of statments:

1. declaration of variables (including arrays and structures) and their data types, and

2. mathematical relations (also called assertions) between the variables.

A variable can take a value belonging to the set specified by its data type. The data type of a variable can either be declared explicitly or be determined from its use. It is immaterial to the specification where the variables are stored, i.e. in main memory or secondary memory. The basic data structure in the language is the array. Variables or structures may be declared to be arrays. A sequential file is considered to be an array of records.

Assertions are essentially equations which define relationships between variables. A variable can have only one value as in mathematics. Each assertion, in fact, defines the value of a variable. The assertions can be composed by the user in any order because they specify relations, which do not have any temporal meaning associated with them.

By the use of free subscripts, a single assertion can define the value of an entire array. Identifiers corresponding to the free subscripts can be declared. The notion and use of free subscripts is similar to that in mathematics. For example, let "F" be an array variable, and "I" a free subscript, then the assertion:

```
        F(I)  =  IF I=1 THEN 1
                           ELSE I*F(I-1)
```

defines the value of the entire array F. Each element of the array F is defined in terms of the previous one, except for the first element which is defined to be 1.

In the above example, the size of the array F is not specified, and hence is possibily infinite. The size can be specified in essentially two ways:

1. An upper bound can be declared for the subscript I; or

2. A special array of the same dimensions and sizes as F can be defined, which

identifies the size for the rightmost dimension of F. This special array is called the *end array*. F is defined upto the largest index such that the corresponding element in the special array is true, and all elements of lower index are false. Its use is illustrated by means of an example:

```
END.F(I)  =  IF I=4 THEN TRUE
                     ELSE FALSE;
```

The above causes an array F to be of size 4. In other words, an array variable F is defined for as many elements until and including the first true element of END.F.

Certain rules govern the usage of subscripts. These have been designed so that the specification can be compiled rather than interpreted. A subscript can occur in one of three forms:

1. a subscript term e.g. I in F(I);
2. an expression of the form (I-k) where k is a positive integer, e.g. (I-1) in F(I-1); and
3. another variable or subscripted *variable, e.g. G(I) in F(G(I)).*

For a subscripted variable which occurs on the left hand side of the equation, its subcripts must be in the first form. This makes the consistency analysis simpler.

The above is the essence of a non-procedural language using mathematical equations. There are many additional features in the Model language to handle file organization, and in the Nopal language for fault isolation in testing of physical systems. A processor for such a language analyzes the specifications for consistency, completeness and non-ambiguity; and if successful, generates a program in a high level language. By consistency we mean that the variables are defined only once, and by completeness that the variables are defined at least once. In the generated program, the variables should be defined before they are referenced. This analysis, which is non-trivial when free subscripts are used, is described with respect to the Nopal system in Chapter 5.

3.3 APPROACHESTO MODULARITY

Need for modularity has been discussed in Chapter 1. As discussed earlier, it means sub-dividing a problem into smaller specifications, and compiling each of the specifications seperately. A number of alternative approaches are possible to achieve modularity in non-procedural languages. Some of them are described below.

The simplest approach is to divide a large specification into smaller specifications which communicate through commonly named variables. The aggregate of sub-parts is exactly equivalent to the total, obtained by simply putting the sub parts together and forming one large specification.

In a different approach, each sub-part represents a specification of a function, and these functions may be used in other sub-parts. In this approach, the functions may be specified once and used many times resulting in a more compact overall specification. A judicious choice of functions may also correspond to a decomposition of the specification at the logical or "conceptual" level.

Still another approach utilizes the idea of data abstaction. In this approach a sub-part specifies an abstract data type and the functions which are allowed to operate on variables of the data type. The data type can now be used in other sub-parts. In other words, variables in other sub-parts can be declared to be of the defined data type and be operated upon by the specified functions. This approach has the advantage similar to the functional approach, namely, that a data type specified once in a sub-part may be used many times in other sub-parts.

A procedure in a programming language accomplishes an action (or performs a sequence of steps). A procedure is used knowing "what" it accomplishes without knowing "how" it accomplishes it, and similarly it is defined knowing "what" it is supposed to accomplish without knowing "how" it will be used. Thus the abstraction separates use from definition and introduces modularity. In a similar fashion, a data type is specified independent of its use. It represents a set of objects which satisfy certain properties, and frequently, these objects correspond to the user's problem domain, e.g. stacks, tokens, sets etc. A variable of the said data type represents one of these objects and allows us to express relationships directly among these objects in a non-procedural specification. Thus it also makes the specification closer to the terminology of the problem.

Another advantage of this approach is that the representation for the variables belonging to abstract types need not be known while writing the specification. This allows the representation of a data type to be modified without affecting its use. The representation of a data type is specified by means of a sub-part defining the data type, and can be changed by changing that sub part alone.

In light of the above advantages, this latter approach to modularity is adopted in this dissertation. (Another motivation for chosing the latter approach is that it provides a convenient way to represent devices for testing in the Nopal system.) It should be recognized that the sub-part specifying the data type allows the funtions which can operate on the data type to be clustered together. The use of this abstraction serves as the guiding principle for clustering of the functions. We hope to illustrate below that this is, indeed, a natural way to modularize non-procedural languages.

3.4 USEOF ABSTRACT DATA TYPES

Each variable in a non-procedural specification has a data type, which gives the set of its possible values. The data type of a variable can be elementary (i.e. one defined by the language e.g. real, integer, character etc.) or can be one of the abstract types.

An abstract data type must be specified non procedurally by means of a specification called the *module* for that data type. Just as a variable of the elementary data type can be operated upon by the functions for the data type, e.g. functions +, -, *, / for the integers, a variable of the abstract data type can only be operated upon by the set of functions defined in the corresponding module.

Variables can occur in assertions as defined below. Assertions define the relationships between the variables. An assertion is of the form:

$$A_1(I_1,\dots I_{d1}) = \tau(I_1,\dots I_{d1}, A_1,\dots A_n)$$

where $A_1,\dots A_n$ are names of array variables. $I_1,\dots I_{d1}$ are subscripts for d1 dimensional array variable A_1, and τ denotes the expression formed using function symbols, subscripts and array variables.

Expressions are formed using notation familiar in mathematics. Informally:

1. An array variable followed by a list of subscript expressions is an expression, and the data type of the variable gives the set to which the value of the expression belongs.

2. A function symbol followed by expressions in parenthesis is an expression. The data types of the expressions should match the domains of the function symbol. The data type of the new expression formed is the range (as in mathematics) of the function. The expression defines a mapping from the domains to the range of the function

3. Symbols +, -, *, / denote the functions for addition, subtraction, multiplication and division; and they may be used as infix operators. Similarly, the function if then else (cond, x, y) with three arguments can be written in its familiar form: if

cond then x else y.

Data types place restrictions on the ways in which expression can be combined to form new expressions. In particular, the data types of the arguments of a function must match the domains of the function.

There is no distinction in the use of elementary and abstract data types. The user of the language once provided with a set of data types and the functions which can be performed on them may use the given set of data types without ever knowing which are elementary and which abstract. The use of the abstract data types, therefore, does not require any new meaning to be given to variables or assertions in the non-procedural language.

3.5 SPECIFICATIONOF ABSTRACT DATA TYPES

This section introduces the concept of a module for the specification of the abstract data types. The specification of an abstract data type is independent of its use. It is specified non-procedurally within the framework of the language intrduced in section 3.2. The module specification can be analysed for inconsistency, incompleteness, and ambiguity, independent of other module specifications. In particular, the variables in the module are single valued, subscripts are consistently used, and are independent of the subscripts and dimensions in other module specifications. Finally, as will be shown later, the generated program supports the use of variables of the defined abstract type in other modules.

A module consists of: (1) a header - which gives the name of the abstract data type, (2) data declarations - which give the representation for the abstract data type, and (3) module-functions (*modfuns* for short) - which specify the functions which can operate on the abstract data type being specified by the module. The function specification consists of assertions, including formal paramaters and return value.

A module named, say ADT, specifies a representation for an abstract data type ADT. By *representation* is meant the components of a data type. The word "representation", rather than the word "data structure", is used because the components themselves can be abstract, in which case they are specified by means of other modules. A modfun may return a value of type ADT, in which case the value is defined by defining the value of variables in the representation. This is done by means of assertions in the body of the modfun. If the value is specified using the formal parameters of the modfun, then the modfun specifies the relationship between the formal parameters and the value returned. If one of the parameters is of type ADT, its representation is accessible in the module ADT and can be used in defining the return value.

In general, the return value of a modfun may be of any arbitrary data type. Appropriate function must be used to define the return value.

3.6 ANEXAMPLE - STACK

The ideas presented in the previous section are illustrated by means of an example in this section. The syntax of assertions has already been explained; the syntax of the declarations is somewhat like Pascal and PL/I. The subscripts are declared by means of a statement of the form:

```
<subs> IS A SUBSCRIPT;
```

where <subs> is the name of the subscript.

The example chosen is stack of integers. It has four modfuns. Their domains and range are:

Emptystack: $\rightarrow$ stack
Push: stack $*$ integer $\rightarrow$ stack
Pop: stack $\rightarrow$ stack
Top: stack $\rightarrow$ integer

The *emptystack* maps from null domain to an empty stack; *Push* maps a stack S and integer X to a stack whose top element is X and the remaining part is the same as stack S; *Pop* maps a stack S to another stack which is the same as S except with the top element removed; and *Top* maps a stack S to an integer X such that X is the same as the top element of S.

The above is an informal description of stack in English. The module STACK gives the formal specification of stack and its functions. The specification captures the concept expressed informally above and makes it precise. (The formal semantics of the module is discussed in Section 3.8.)

Consider the following example having an array A of stacks. Each of the elements of the array A is a stack onto which integers from arrays P and Q are pushed.

```
MAIN EX1;
   DCL A:STACK ARRAY(10),
       P,Q:INTEGER ARRAY (10);
       R:BOOLEAN ARRAY (10);

   I IS A SUBSCRIPT;
   A(I)  =  IF I=1 THEN EMPTYSTACK
              ELSE IF R(I) THEN PUSH(A(I-1),P(I-1));
                ELSE PUSH(A(I-1),Q(I-1));
             /*  ARRAYS P,Q,R ARE ASSUMED TO BE DEFINED ALREADY. */
END EX1;
```

The STACK module is:

```
MODULE STACK;
      DCL  1  STACK: RECORD,
             2 TOPZ: INTEGER,
             2 Z: INTEGER ARRAY(100);
      J IS A SUBSCRIPT;

MODFUN PUSH(S:STACK, X:INTEGER) RETURNS (S1:STACK);
      S1.TOPZ = S.TOPZ + 1;
      S1.Z(J)=IF J<=S.TOPZ THEN S.Z(J)
                               ELSE X;
      END.S1.Z(J) = (J=S1.TOPZ)
END;
MODFUN POP(S:STACK) RETURNS (S1:STACK);
      S1.TOPZ = S.TOPZ-1;
      S1.Z(J) = S.Z(J);
      END.S1.Z(J) = (J=S1.TOPZ);
END;
MODFUN TOP(S:STACK) RETURNS (X:INTEGER);
      X=S.Z(S.TOPZ);
END;
MODFUN EMPTYSTACK RETURNS (S1:STACK);
      S1.TOPZ=0;
END;
END STACK;
```

In the above example, representation for a stack consists of two components: a 100 element integer array called Z, and an integer TOPZ. The familiar notation of record (as in PL/I, Pascal etc.) is used to show the components of stack. Variables S1 and S which occur in the modfuns are of data type stack. Outside the STACK module the two components of stack are not visible, however, inside the module the variables S and S1 are seen to consist of two components. To refer to their components qualified names are used, e.g. S1.TOPZ refers to a component of stack S1, while S.TOPZ refers to that of stack S.

The STACK module can be analysed for consistency independent of the use of stack data type. The modfuns PUSH and POP define a stack by defining the value of its components which satisfy certain relationship with components of another stack. For example, the modfun PUSH defines the value of a stack S1 in terms of stack S and integer X, which are the formal parameters of PUSH. The two components, Z and TOPZ, of the stack S1 are defined in terms of the components of the stack S and integer X. EMPTYSTACK defines a stack which satisfies certain properties independent of any other stack; TOP defines an integer with respect to the stack S, which is a again a formal parameter.

The definition of the array of stacks, A, in the main module does not require knowledge of the representation of stack. It can be analysed for consistency independent of the module STACK.

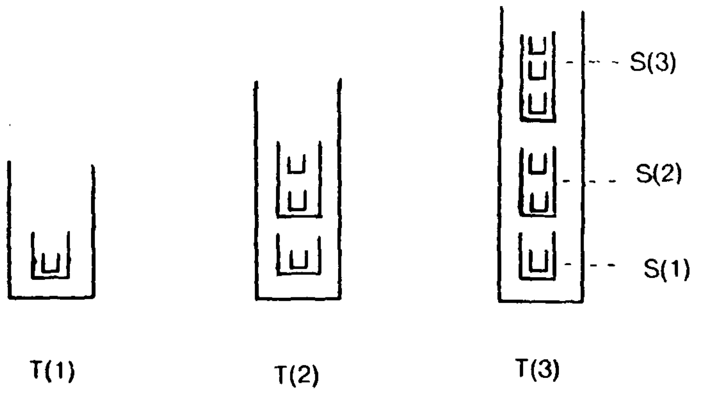
3.7 RECURSIVE DEFINITIONS

Modules can be used to define data types whose representation is specified recursively. For an abstract data type, say ADT, its *representation* can be specified in terms of variables which themselves can be of type ADT. Modfuns can now be applied to these variables recursively to define their values.

The recursive data types are illustrated below by means of an example. Stack-of-stacks data types (SOS for short) is chosen to show the similarity with the previous stack example. The specification of SOS is same as that for STACK except that the data type of the array Z in the representation of SOS is of type SOS instead of INTEGER.

```
MAIN EX2;
      DCL S,T: SOS ARRAY(100);

      S(I) = IF I=1 THEN PUSHS(EMPTYSOS, EMPTYSOS)
                     ELSE PUSHS(S(I-1), EMPTYSOS);
      T(I) = IF I=1 THEN PUSHS(EMPTYSOS, S(1)),
                     ELSE PUSHS(T(I-1),S(I));
END EX2;
```



⊔	= stack symbol
⊔	= stack containing "-"

Figure 3-1: EXAMPLE EX2: USING STACK OF STACKS

```
MODULE SOS;
       DCL 1 SOS: RECORD,
             2  TOPZ: INTEGER,
             2     Z: SOS ARRAY(100);
       J IS A SUBSCRIPT;
MODFUN PUSHS(S:SOS,X:SOS) RETURNS (S1:SOS);
       S1.TOPZ = S.TOPZ + 1;
       S1.Z(J) = IF (J <= S.TOPZ) THEN S.Z(J)
                                  ELSE X;
       END.S1.Z(J) = (J = S1.TOPZ);
END;
MODFUN POPS(S:SOS) RETURNS (S1:SOS);
       S1.TOPZ = S.TOP - 1;
       S1.Z(J) = S.Z(J);
       END.S1.Z(J)  =  (J = S1.TOPZ);
END;
MODFUN TOPS(S:SOS) RETURNS (X:SOS);
       X = S.Z(S.TOPZ);
END;
MODFUN EMPTYSOS RETURNS (S1:SOS);
       S1.TOPZ = 0;
END;
END SOS;
```

Stack of stacks (SOS), as defined above, is not very useful because it cannot handle a stack of integers. A SOS can only contain other SOS's. The difficulty arises because the data type of Z, a component of SOS, is restricted to be of data type SOS; hence it does not allow a stack of integers to be part of SOS. There are a number of ways of dealing with the problem, e.g. parameterized modules, disjoint union of data types etc. explained below. A particularly elegant method is by using parameterized modules.

A *generic* or *parameterized* module defines a class of data types. Different values of the parameter of the module result in different members of the class of data types. The SOS example is rewritten using generic module. A single module defines stack of integers, stack of characters, stack of stacks etc. depending on the value of the parameter.

STK specifies a parameterized stack. Its parameter is a data type which determines the components which a given STK can have. For example, S is declared to be an array of stacks of integers i.e. each of the element of the array S is a stack and can contain integers. Similarly, T is an array of stack of stacks. At the time of declaration of variables of data type STK, the parameter of STK must be specified.

```
MAIN EX3;
      DCL S: STK[INTEGER] ARRAY(100),
          T: STK[STK] ARRAY(100);

      S(I) = PUSHSTK(EMPTYSTK(INTEGER), I);

      T(I) = IF I=1 THEN PUSHSTK(EMPTYSTK(STK), S(1)),
                     ELSE PUSHSTK(T(I-1),S(I));
END EX3;
```

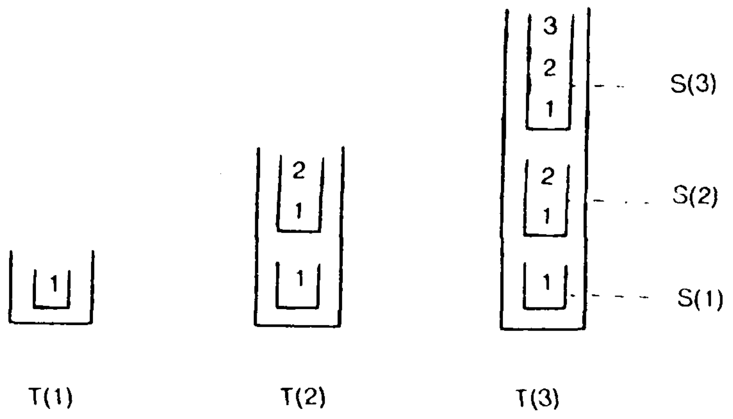


Figure 3-2: EXAMPLE EX3: USING PARAMETERIZED STACK

Figure 3-2 illustrates the various stacks in EX3, by means of a picture.

```
MODULE STK[U: TYPE];
       DCL 1 STK: RECORD,
             2  TOPZ: INTEGER,
             2     Z: U ARRAY(100);
       J IS A SUBSCRIPT;
MODFUN PUSHSTK(S:STK[V], X:V) RETURNS (S1:STK[V]);
       S1.TOPZ = S.TOPZ + 1;
       S1.Z(J) = IF (J <= S.TOPZ) THEN S.Z(J)
                                  ELSE X;
       END.S1.Z(J) = (J = S1.TOPZ);
END;
MODFUN POPSTK(S:STK[V]) RETURNS (S1:STK[V]);
       S1.TOPZ = S.TOP - 1;
       S1.Z(J) = S.Z(J);
       END.S1.Z(J)  =  (J = S1.TOPZ);
END;
MODFUN TOPSTK(S:STK[V]) RETURNS (X:V);
       X = S.Z(S.TOPZ);
END;
MODFUN EMPTYSTK(V:TYPE) RETURNS (S1:STK[V]);
       S1.TOPZ = 0;
END;
END STK;
```

The construct of disjoint union also allows a single module to define a class of data types. A variable is said to be of disjoint union of data types X and Y, if the variable can take a value denoted by either of the data types, and there is a way to distinguish whether its value is of data type X or data type Y. Part of SOS example is rewritten below to illustrate the idea. In the example, a tag field is associated with the SOS record, which indicates one of two possible choices in the variant part of the record. Thus depending on the tag field, it represents a stack of integers or stack of stacks. (It is assumed that TYPE OF STACK is a data type defined to be a set consisting of two keywords INTEGER and SOS. CASE has similar meaning as in Pascal.) Accordingly, the data type of the parameter X in the function PUSHSTK is of type SOS or INTEGER.

```
MODULE STK;
      DCL 1 STK: RECORD,
            CASE TAG: TYPE-OF-STACK OF
            INT: 2 TOPZI: INTEGER,
                 2    ZI: INTEGER ARRAY(100),
            SOS: 2 TOPZS: INTEGER,
                 2    ZS: STK ARRAY(100);
      J IS A SUBSCRIPT;
MODFUN PUSHSTK(S:STK, X:CASE(STK,INTEGER)) RETURNS(S1:STK);
 CASE S.TAG OF
 INT:   S1.TOPZI = S.TOPZI + 1;
        S1.ZI(J) = IF (J <= S.TOPZI) THEN S.ZI(J)
                                     ELSE X;
        END.S1.ZI(J) = (J = S1.TOPZI);
 SOS:   S1.TOPZS = S.TOPZS + 1;
        S1.ZS(J) = IF (J <= S.TOPZS) THEN S.ZS(J)
                                     ELSE X;
        END.S1.ZS(J) = (J = S1.TOPZS);

END;

      .
      .
      .

END STK;
```

Unlike the parameterized module, the class of stacks that STK specifies is limited to those explicitly defined in the module, e.g. in the above example it is limited to two INTEGER and STK. In case of the parameterized module, the class of stacks specified by STK is left open in the specification.

Disjoint union and parameterization are not included while defining the semantics of modules to keep the treatment simple. Parameterized modules (or disjoint union) can always be replaced by a number of different modules, each corresponding to a different value of the parameter (or a different data type in the union).

3.8 SEMANTICS OF MODULES

This section defines the denotational semantics or the fix point semantics of the modules. The denotational approach has been chosen because the semantics so defined is independent of the computation rules (or the interpreter) used to evaluate the modules. This may be contrasted with the operational or axiomatic approach, in which the semantics is defined in terms of the interpreter. The denotational approach is particularly suited for non-procedural languages, because these languages are independent of the sequence of control of the statements. The denotational semantics of modules shows two things: (1) the module defines a set of functions, and (2) the functions can be computed.

The equations and arrays in the specification are considered as partially defined recursive functions. This allows us to translate our notation into the standard recursive function equations, and use the results regarding least fix point already known in that domain.

Some of the important definitions used in denotational semantics are described here. Partial ordering "$\preceq$" on every extended domain $D^+ = D \cup \{\perp\}$, where $\perp$ stands for the undefined value, corresponds to the notion of *less defined than or equal to*. It is defined as:

$$\perp \preceq d, \text{ and } d \preceq d \;\; \forall d \in D^+$$

A function f is said to be monotonic if:

$$x \preceq y \;\Rightarrow\; f(x) \preceq f(y) \;\; \forall x,y \in D^+$$

Starting with these basic definitions semantics for recursive equations is defined (Chap. 5 in [37]).

First, the semantics of equational specification (Section 3.2) is presented. It is based on [40]. Later, it is extended to give semantics to modules. It is also shown that a module specification defines a set of algebraic axioms satisfied by the abstract data type.

An equational specification, introduced informally in Section 3.2, for the array symbols A_1, ... A_n of dimensionalities d_1, ... d_n, and data types T_1, ... T_n respectively, is a system of equations:

$$A_1(I_1, \dots I_{d_1}) = \tau_1(I_1, \dots I_{d_1}, A_1, \dots A_n)$$

$$\vdots$$

$$A_n(I_1, \dots I_{d_n}) = \tau_n(I_1 \dots I_{d_n}, A_1, \dots A_n)$$

The terms $\tau_i(I_1, \dots I_{d_i}, A_1, \dots A_n)$ for i = 1 to n are defined recursively as follows:

Letters $f_1, f_2, \dots$ are used to denote functions over array values; and $g_1, g_2, \dots$ are used to denote integer valued functions used as subscripts. A subscript is defined as follows:

1. I_k is a subscript. Its appearance in τ_i satisfies $k \leq d_i$.

2. I_k-c is a subscript, where c is an interpreted integer constant.

3. If J_1, ... J_n are subscripts, then so is $g_i(J_1, \dots J_n)$.

A term is defined as:

1. If J_1, ... J_n are subscripts, then $A_i(J_1, \dots J_n)$ is a term of data type T_i.

2. If t_1, ... t_m are terms of data types S_{i1}, ... S_{im} respectively, then $f_i(t_1, \dots t_m)$ is a term of data type S_i (where occurrance of the symbol f_i is always followed by terms of the data types S_{i1}, ... S_{im}).

An interpretation for a specification consists of

1. domains D_1, ... D_r over which the elements of array vary, and let $D = \{D_1, \dots D_r\}$;

2. a one-to-one onto mapping M such that: $M(x) = D_i$, where x is a data type, and $D_i \epsilon D$;

3. an assignment of concrete functions to the symbols $\{f_i\}$, i.e. $I[f_i]: D_{i_1}{}^+ \times D_{i_2}{}^+ \times \dots D_{i_m}{}^+ \to D_i{}^+$ where m is the arity of f_i, S_{i_j} the data type for the j^{th} argument of the function satisfies the relation $M(S_{i_j}) = D_{i_j}$, and similarly the data type for the range of the function satisfies $M(S_i) = D_i$, where $D_{i_j} \epsilon D$ for $1 \leq j \leq m$, $D_i \epsilon D$; and

4. an assignment of concrete natural number functions to the symbols $\{g_i\}$ i.e. $I[g_i]$: $(Z^+)^{d_i} \rightarrow Z^+$, where d_i is the arity of g_i.

where D_i^+ is the extended domain, $D_i^+ = D_i \cup \{\perp\}$; and Z^+ is the extended domain of natural numbers, $Z^+ = Z \cup \{\perp\}$, where $\perp$ stands for undefined value. Moreover, f_i and g_i are restricted to be monotonic in the sense of partial ordering.

Least fix point semantics is adopted to give a meaning to the specification. Thus, the solution to a given set of equations is taken to be the least fix point solution. Each of the A_i is specified as a partial function, A_i: $Z^{d_i} \rightarrow D_i$. Monotonicity of functions f_1, f_2, ... g_1, g_2, ... assures that the τ_i are continuous. Therefore, the least fix point solution exists and is unique (Thms. 5-1, 5-2 in [37]).

The semantics of module specification is presented next. A module specification consists of the declaration of the representation for the abstract data type specified by the module, and the operations which may be performed on variables of the type. Let an abstract data type called ADT be specified by a module of the same name. The modfuns specified by the module may be divided into two classes:

1. those which return a variable of data type ADT, and

2. those which return a variable of data type other than ADT.

Semantics of modfuns for each of the classes will be presented.

Representation of the abstract data type ADT, in its most general form, is given by a structure of the form:

```
dcl 1 ADT: record,
       2 A_1: T_1,
       .
       .
       .
       2 A_n: T_n;
```

where A_i's are the variables and T_i's are their data types respectively. Each of the A_i's may be arrays or simple variables. A structure declaration is given the semantics of a tuple, and therefore, the structure for ADT denotes the tuple:

$\langle A_1, A_2, \dots A_n \rangle$.

Note that since any of the T_i's may in turn be of type ADT, the abstract data type, ADT, may be defined recursively.

A modfun in the module ADT which returns a variable of data type ADT is of the form:

```
MODFUN OPC($C_1$:ADT, ... $C_p$:ADT,$B_1$:$u_1$, ... $B_q$:$u_q$) RETURNS(C:ADT);
  $C.A_1(I_1, \dots I_{d_1}) = \tau_1(I_1, \dots I_{d_1}, A_1, \dots A_n, \underline{C}, \underline{B}, \underline{op})$
   .
   .
   .
  $C.A_n(I_1, \dots I_{d_n}) = \tau_n(I_1, \dots I_{d_n}, A_1, \dots A_n, \underline{C}, \underline{B}, \underline{op})$
END
```

where OPC is the name of the modfun; C_i's are the formal parameters of data type ADT; B_i's are formal parameters of data types u_i's respectively where none of the u_i's is ADT; and τ_i's represent expressions. $\underline{C}$ is used to denote $C_1, C_2, \dots C_p$; $\underline{B}$ to denote $B_1, B_2, \dots B_q$; and $\underline{op}$ to denote the modfuns in the module. A_i's are array symbols and are components of the tuple of ADT, defined earlier. The d_i's give the arities of the corresponding A_i's, and I_j's are the subscripts of the respective A_i's, where $1 \leq j \leq d_i$.

The expressions τ_i's can now be defined as below. Letters $f_1, f_2, \dots$ are used to denote functions over array values; and $g_1, g_2, \dots$ are used to denote integer valued functions used as subscripts. A subscript is defined as follows:

1. I_{d_i} is a subscript. Its appearance in τ_i satisfies $k \leq d_i$.

2. I_{d_i}-c is a subscript, where c is an interpreted integer constant.

3. If $J_1, \dots J_n$ are subscripts, so is $g_i(J_1, \dots J_n)$.

A term is defined as:

1. If $J_1, \dots J_{d_i}$ are subscripts, then $C_j.A_i(J_1, \dots J_{d_i})$ is a term of data type T_i; where j satisfies $1 \leq j \leq p$, and i satisfies $1 \leq i \leq n$.
2. $B_i(I_1, \dots I_{d_i})$ is a term of data type u_i, where $1 \leq i \leq q$.
3. If $t_1, \dots t_m$ are terms of data types $S_{i1}, \dots S_{im}$ respectively, then $f_i(t_1, \dots t_m)$ is a term of data type S_i, where occurrance of f_i is always followed by terms of data types $S_{i1}, \dots S_{im}$.
4. Same as (3) with the symbol f replaced by op, where op ε <u>op</u>, and S_{ij}'s replaced appropriately.

The following interpretation is given to the above set of equations.

1. a set of basic domains $D_1, \dots D_r$ including domain Z of the set of positive integers, and let $D_{tot} = \{D_1, \dots D_r, D\}$, where set D is defined by the module;
2. for each of the data types T_i's, u_i's and ADT define a one-to-one onto mapping M such that:

   $$M(ADT) = D$$
   $$M(x) = d \quad \text{where } x \;\varepsilon\; \{T_1 \dots T_n, u_1 \dots u_q\}$$
   $$\text{and } d \;\varepsilon\; \{D_{tot} - D\}$$

3. for each symbol f_i of arity d_i assign a concrete function:

   $$f_i: D_{i1} \times \dots \times D_{id_i} \rightarrow D_i$$
   $$\text{where } D_i \;\varepsilon\; D_{tot}$$
   $$\text{and } \forall k,\ D_{ik} \;\varepsilon\; D_{tot}$$

   where S_{ij}, the data type for the j th argument of the function f_i satisfies $M(S_{ij}) = D_{ij}$, and S_i, the data type for the range of the function satisfies $M(S_i) = D_i$;

4. for each symbol g_i of arity d_i assign a concrete number theoretic function:

   $$g_i: (Z^+)^{d_i} \rightarrow Z^+;$$

5. a set of projection functions P_i's such that

   $$\langle A_1, \dots A_n \rangle.A_i = P_i(\langle A_1, \dots A_n \rangle)$$
   and with subscripts and symbol C_j for the tuple
   $$C_j.A_i(I_1, \dots I_{d_i}) = P_i(C_j)(I_1, \dots I_{d_i})$$

for $1 \leq j \leq p$
and $1 \leq i \leq n$;

6. a set of functions OPC_1, ... OPC_n (instead of the multi-valued function OPC) defined as follows:

$$OPC_1(\underline{C},\underline{B},I_1,...I_{d_1}) = C.A(I_1,...I_{d_1})$$
$$\vdots$$
$$OPC_n(\underline{C},\underline{B},I_1,...I_{d_n}) = C.A(I_1,...I_{d_n});$$

where $\underline{C}$ and $\underline{B}$ are the formal oarameters of OPC.

With the interpretations (5) and (6) the equational specification of a module can be written in the familiar form of recursive equations:

$$OPC_1(\underline{C},\underline{B},I_1, ... I_{d_1}) = \tau_1(I_1, ... I_{d_1},\underline{P},\underline{C},\underline{B},\underline{op})$$
$$\vdots$$
$$OPC_n(\underline{C},\underline{B},I_1, ... I_{d_n}) = \tau_n(I_1, ... I_{d_n},\underline{P},\underline{C},\underline{B},\underline{op})$$

where $\underline{op}$ is the set of operations with proper substitutions. (For example. $OPC \in \underline{op}$ is written as the tuple $\langle OPC_1, ... OPC_n \rangle$.) $\underline{P}$ represents $P_1, ... P_n$.

For each of the modfuns of class 1, i.e. those which return a value of data type ADT, a similar set of recursive equations can be written.

For each of the modfuns of class 2, i.e. those which return a value of data type other than ADT, a similar but simpler set of recursive equations can be written. A modfun belonging to the second class is of the form:

MODFUN $OPD(C_1$:ADT, ... C_p:ADT,B_1:u_1, ... B_q:u_q) *RETURNS*(E:u)
  $E(I_1, ... I_{d_1}) = \tau_1(I_1, ... I_{d_1},A_1, ... A_n,\underline{C},\underline{B},\underline{op})$
END

which reduces to the recursive equation:

$OPD(\underline{C},\underline{B},I_1, \ldots I_d) = \tau(I_1, \ldots I_d,\underline{P},\underline{C},\underline{B},\underline{op})$

All the recursive equations are now put together, by renaming the variables which occur as formal parameters of modfuns, to avoid clash of names.

The functions $f_1,f_2, \ldots, g_1,g_2, \ldots$ are constrained to be monotonic in the sense of partial ordering. The projection functions are monotonic because:

Let P_i be the i^{th} projection function. Now
let $x = \langle x_1, \ldots x_i, \ldots x_n \rangle$
and $y = \langle y_1, \ldots y_i, \ldots y_n \rangle$

$x \preceq y \Rightarrow x_i \preceq y_i$ for all i, (where "$\preceq$" stands for less defined than or equal to)
$P_i(x) = x_i$
$P_i(y) = y_i$
$x \preceq y \Rightarrow P_i(x) \preceq P_i(y)$
therefore, P_i is monotonic.

Hence, τ_i's are continuous and the least fix point solution of the recursive equations exists (Thms. 5-1, 5-2 in [37]).

The set D, which corresponds to the data type ADT, is defined inductively as follows:

1. *Base step:* For a modfun OP which returns a value of data type ADT, and none of whose formal parameters is of type ADT, the tuple defined by OP: $\langle OP_1, \ldots OP_n \rangle(\underline{B})$ is a member of set D. $\underline{B}$ are the formal parameters of function OP, and OP represents a tuple of functions.

2. *Inductive step:* For a modfun OPC which returns a value of data type ADT, the tuple defined by the modfun: $\langle OPC_1, \ldots OPC_n \rangle(\underline{C},\underline{B})$ is a member of set D; where $\underline{C}$ are the members of set D, and $\underline{B}$ are members of other domains ($D_{tot} - D$).

The existence of the least fix point solution assures the existence of the set D.

With the semantics of the modules defined, algebraic axioms about the abstract data types can now be proved. The proof involves substituting non-procedural equations for the occurrances of the module functions, and reducing the equations until the desired equality is obtained. This is illustrated by means of the STACK example. Note that since it does not involve recursive definition of the data type the derivation is straight forward.

To prove. POP(PUSH(S.X)) = S. where S is a stack. and X is an integer.

Proof:

LHS

= POP(S') where S' is a stack and

$$S'.TOPZ = S.TOPZ + 1$$

$$S'.Z(J) = \begin{cases} S.Z(J) & \text{if } J \leq S.TOPZ \\ X & \text{if } J = S'.TOPZ \\ \perp & \text{otherwise} \end{cases}$$

= S'' where S'' is a stack and

$$S''.TOPZ = S'.TOPZ - 1$$

$$S''.Z(J) = \begin{cases} S'.Z(J) & \text{if } J \leq S'.TOPZ \\ \perp & \text{otherwise} \end{cases}$$

= S'' where $S''.TOPZ = S.TOPZ$

$$S''.Z(J) = \begin{cases} S.Z(J) & \text{if } J \leq S.TOPZ \\ \perp & \text{otherwise} \end{cases}$$

= S'' where $S''.TOPZ = S.TOPZ$

$S''.Z(J) = S.Z(J)\ \forall J$ (from Lemma 1.)

= S

Q.E.D.

Lemma 1: $\forall S \in$ STACK, $S.Z(J) = \perp$ for $J > S.TOPZ$

Proof: Since the stacks can only be defined by the modfuns in the module STACK, the proof follows from induction:

Base step: Follows from the definition of the EMPTYSTACK.

Induction step: Let S be a stack satisfying the proposition of the Lemma.

Claim: The stacks POP(S,X) and PUSH(S,X) also satisfy the lemma.

Proof:

1.

Let S' = POP(S,X)
S'.Z(J) = $\perp$ for J > (S.TOPZ - 1)
= $\perp$ for j > S'.TOPZ

2.

Let S' = PUSH(S,X)
S'.Z(J) = $\perp$ for J > S'.TOPZ

Q.E.D.

3.9 SUMMARY

This chapter introduces a non-procedural language based on equations. Use of abstract data types has been proposed as a means to introduce modularity in the non-procedural language. It has been argued that the use of abstract data types is consistent with the philosophy of non-proceduralness, and leads to modular specifications.

The notion of "module" has been introduced to allow specification of the abstract data types. It allows the definition of the representation of the abstract data types, and the specification of the functions which can operate on it. These functions are specified non-procedurally by means of equations.

Finally, the denotational semantics of the modules is defined. It is shown that an abstract data type defined by a module is a well defined set. It is also illustrated that the axioms satisfied by the abstract data types can be derived from the equational specification.

Chapter Four

THE NOPAL LANGUAGE

4.1 OVERVIEW OF THE NOPAL LANGUAGE

Nopal is a descriptive language used to write specifications for the programming of automotic test systems. It can be used for testing of electronic circuits, mechanical systems, chemical processes etc. It also has the capablity to perform general purpose computational tasks.

Basic statements in Nopal are assertions and data declarations similar to those described in Chapter 3. However, Nopal has additional constructs which are superimposed on the assertions and data declarations. These additional features facilitate the specification of testing. The most important construct is that of a test. A test section consists of a specification of a physical test. Outcome of the test, i.e. passing or failing the test, determines fault isolation. There are also sections to describe the UUT (Unit Under Test) and the ATE (Automatic Test Equipment). These sections are needed to check consistency of interfaces with the UUT and ATE.

Several features of Nopal are extremly important in providing ease of use. First, the language is non-procedural. The user saves effort because the execution order of events or control logic need not be specified. Second, the specification can be divided into sub-parts, the *modules*. Each of the modules can be specified and processed by the language processor independently. This is the essence of modularity of a specification. Third, each of the modules may be further divided into *data declarations* and *functions*. The functions are divided into *tests*, *diagnoses* and *messages*. Each test has sub-parts: *stimulus*, *measurement* and *logic*. All these correspond to notions which occur in testing. Fourth, the language allows incremental development of specification. Tests can be added to a specification without changing the tests already specified.

The Nopal system produces a number of reports which serve as the documentation for the specification. It also enhances the user-system interaction, and helps the user in locating errors in the specification.

In this chapter, the Nopal language is described informally with examples. A more detailed explanation including the formal syntax is given in [46].

A *Nopal specification* gives a complete description of the desired tests specific to a given UUT and ATE. In general, a Nopal specification consists of a collection of *modules*. One of the modules is called the *main module* and it consists of the tests on a given UUT with an ATE. Communication between the modules is by means of abstract data types. A module (except the main module) represents an abstract data type which can be used by other modules.

A module specification includes the data representation for an abstract data type together with the functions (called module functions or *modfuns* for short) which can operate on the variables of the abstract type (also called *abstract variables* for short). An abstract data type that has been specified by means of a module can be used in any of the modules. The abstract variables are defined and operated upon by means of the modfuns specified in the module.

The modules are specified non-procedurally. For organizational purposes each module can be divided into four major sections, which can be given in any order. They are:

1. Data declaration specification,

2. Modfun specification,
3. UUT specification, and
4. ATE specification.

Each of the four sections are explained briefly below, followed by a more detailed description later.

The data declaration specification provides the data types of the variables and the data structure used in the specification.

The modfun specification describes the mapping between the input and the output parameters of the modfun. The main module has only one (implicit) modfun, while the other modules may have more than one. Each modfun consists of *tests, diagnoses and messages*. The tests may be further sub-divided into *stimuli, measurements and logic*. A test corresponds to the notion of a physical test on the UUT, i.e. application of stimuli, taking of measurements and selection of diagnoses, based on the results of the test as expressed in the logic part. The diagnoses report of the test consists of messages that typically identify the faults in the UUT.

The UUT specification gives the description of failure modes, connection points etc. of the UUT. This description is cross-checked by the language processor for consistency within the module.

The ATE specification provides the description of the functions used in the module. These functions can be used for application of stimuli, taking of measurements, or for computations. These functions must be specified outside the module. They can be either part of a library of functions, or they can be specified as modfuns by other modules. The ATE specification gives the function parameters and their data types. In other words, it gives the specification of the interface with the rest of the modules and with ATE.

4.2 DATA DECLARATION SPECIFICATION

The data declaration specification allows the user to declare the data types of variables. The data type of a variable specifies the set to which (the value of) the variable must belong, and the operations which can be performed on it .

Data types can be either elementary, e.g. real, integer, or character or they can be abstract, in which case they must be specified by means of modules.

Data declarations include specification of the structure of the data. The two basic structuring methods are: (1) arrays, and (2) structures An array is a homogeneous structure of elements, all of which are of the same data type. A structure, on the other hand may consist of components of different types which are grouped together. The components themselves can be arrays or structures, thus permitting structures of arbitrary complexity to be declared.

A structure may be viewed as a tree. The root of a tree represents the entire structure, and its descendents correspond to the components of the structure. Finally, the leaves of the tree correspond to the individual variables in the structure. Below are some examples of declarations of variables:

```
DCL   A,B,X : INTEGER;
DCL     Y,Z : STACK ARRAY (10);
DCL     1 P : GROUP ARRAY (5),
          2 Q : GROUP ARRAY (*),
            3 R : INTEGER,
            3 S : REAL;
```

In the first statement, variables A, B and X are declared to be of type integer; in the second, Y and Z are declared to be one dimensional arrays of size 10, and data type stack, and in the third, a three level tree structure is declared. In the tree structure the root is the variable P having the descendent Q which has variables R and S. P is an array of five elements and Q an array of size which is to be specified elsewhere.

A declared structure is implied to be on secondary storage if the data type of the root node is FILE. Name of the root node, in that case, gives the name of a file and the structure declaration gives the structure of the file. In other words, the declared structure represents a file, and is called a *file structure*. For example:

```
DCL  1 F: FILE,
           2 P: GROUP ARRAY (*),
              3 Q: RECORD ARRAY (10),
                 4 R: INTEGER,
                 4 S: REAL,
              3 Q1: RECORD ARRAY (5),
                 4 R1: CHAR;
```

A file F is declared to contain an array P of indefinite size. Each element of P contains 10 Q's and 5 Q1's.

A file structure is considered by the system to be *input* if all the fields (i.e. leaves) of the structure are not defined in the specification, and *output* otherwise. The non-leaf nodes of a file structure can be of type RECORD or GROUP. A non-leaf node which corresponds to a unit of input output on the secondary storage is declared as a RECORD, and GROUP otherwise.

Variables in an input file structure are defined in the generated program by means of a special function called ACCESS. Calls on this function are generated at appropiate places in the generated program for the specification. SAVE function is the exact dual of above for an output file structure. Use of ACCESS and SAVE function is implicit and need not be specified by the user.

For ISAM files, a key is represented by a variable name which is the name of the record prefixed by "PTR." For example, an instance of a record named "Z" in an ISAM file can be defined by means of its key "PTR.Z". The value of the key is passed as a parameter to the ACCESS and SAVE function.

The notion of file structure has been generalized to abstract structures in NOPAL. An input or output structure can be declared to be of abstract type by specifying the data type of the root node as abstract (instead of the keywords RECORD, GROUP or FILE). An example of an abstract structure is:

```
DCL 1  P: AT1  ARRAY (*),
         2  Q: INTEGER,
         2  R: REAL;
```

in which a structure P is declared to be of abstract type AT1.

An abstract structure is considered input if the value of all its fields is not defined in the specification, and output otherwise. Value of input abstract structures is defined in the generated program by means of function named: "ACCESS_" suffixed by the name of the abstract data type. In the previous example, if the structure P is input, its value would be defined by ACCESS_AT1. The ACCESS function for an abstract data type must be specified in its module. Calls to this function are generated at appropriate places in the generated program. An exact dual of the above is the SAVE function for output abstract structures.

Abstract structures allow convenient representation of those files whose physical organization is different from that specified in the main module.

Parameters can be associated with ACCESS and SAVE functions associated with abstract structures. The use of parameters provides a means of communication between the main module and the module which defines the abstract structure. It allows the use of abstract structures to represent testing devices as well. For example, a device which measures ratio of two voltages on two ports can simply be declared as:

```
        DCL 1 GD: GAIN_DEVICE ARRAY (*),
                2 GAIN: REAL;
```

The function ACCESS_GAIN_DEVICE in the module GAIN_DEVICE can give the specification for the appropriate measurements. Thus, each value of the variable GAIN defined by means of the ACCESS function represents a different measurement. Information relating to the ports and ranges can be passed as parameters.

The parameters are specified by a syntax similar to that used for specifying key for ISAM files. In the example above, the parameters of the abstract record GD are given by means of variables named PTR1_GD, PTR2_GD, etc.

4.3 MODFUN SPECIFICATION

This section describes the specification of the module functions (modfuns). Each modfun, like a mathematical function, specifies a mapping from its domain to the ranges. A modfun has zero or more parameters. Parameters are called *source parameters* if their value is defined outside the modfun, and are called *target parameters* if they are defined in the body of the modfun. A modfun can return a value by means of its target parameters or explicitly as in programming or mathematics. The data types of the source parameters are the *domains*, the data types of the target parameters and explicitly returned value are the *ranges* of the mapping specified by a modfun.

The main module has only one implicit modfun; the other modules normally contain more than one modfun. A modfun has four parts:

1. header,
2. test specification,
3. diagnoses, and
4. messages.

The header must be the first statement, after which the tests, diagnoses and messages may occur in any order.

4.3.1 HEADER

Each modfun starts with a header consisting of the keyword MODFUN followed by the name of the modfun, the list of formal parameters and their data types, and the data type of the value explicitly returned by the modfun. It also states which of the parameters are source and which are target. In effect, the header defines the interface with the other modules which use the modfun.

For example the following header:

```
MODFUN  PUSH (S0:S STACK, X:S INTEGER)
             RETURNS (S1:STACK);
```

defines a function called PUSH which has two source *parameters* S0 and X, and it returns a value S1, explicitly. The data type of S0 and S1 are STACK, and the data type of X is INTEGER. Consequently, PUSH specifies a mapping from its domains of STACK and INTEGER to its range STACK.

4.3.2 TEST SPECIFICATION

The test specification consists of a collection of tests. As mentioned earlier, tests correspond to the idea of a physical test on a UUT. A test consists of three parts: 1) *stimuli* that are to be applied to the UUT at the test time, 2) *measurements* that need to be taken and conditions that must be met, and 3) *logic* to select the diagnoses based on the result of passing or failing the test.

Stimuli and measurements both optionally contain two parts, a *conjunction* and a set of *assertions* (Generic word, *waveform*, is used to refer to either a conjunction or an assertion).

A conjunction in stimuli specifies the simultaneous application of stimuli to the UUT, while in the measurement it specifies the simultaneous measurement to be taken of the UUT. All the functions specified in conjunctions must be performed in parallel. For example, the following conjunction:

```
STIM;
    CONJ:   <J1,J2> = PSUPPLY (30V) &
            <J3,J4> = FSOURCE(1KHZ,10V);
```

specifies applying a power supply of 30 volts across the connecting pins J1 and J2, and applying a frequency source of 1kHz and 10 volts between pins J3 and J4.

A conjunction can also be used with an if-statement, in which case it is called an *if-conjunction*. An if-conjunction consists of a boolean condition followed by a conjunction after "THEN" and a conjunction after "ELSE". One of the conjunctions following the "THEN" or "ELSE" part is performed depending on the boolean condition. For example,

```
STIM;
    CONJ: IF VAR<20 THEN <J1,J2> = PSUPPLY (30V)
                    ELSE <J3,J4> = FSOURCE (1KHZ,10V);
```

If a variable VAR is less than 20 then the power supply and otherwise a frequency source is applied.

Conjunctions are used to specify some actions - stimuli or measurements - on the UUT. Assertions, on the other hand, are used to specify relations that must be satisfied by the variables. An assertion specifies relations between variables. It can be used in two roles: as an explicit definition of variables or to specify a condition on the variables. Variables defined in an assertion are said to be *target* variables of the assertion. All others variables in the assertion are called *source* variables of the assertion.

If an assertion does not have any target variable then it specifies a relation which is tested for truth value. An assertion evaluates to *true* if the specified relation is satisfied, otherwise it evaluates to *false*. Assertions which have target variable(s), are always taken to evaluate to true.

The syntax of assertions is:

```
ASRT: <EXPRESSION1> <RELATIONAL OPERATOR> <EXPRESSION2>
                              SOURCE: <LIST OF VARS>
                              TARGET: <LIST OF VARS>;
```

<expression1> and <expression2> are arithmetic or boolean expressions. <relational operator> is one of (= ,<,>,< = ,> = , ≠). <list of vars> is a list of variables with their subscript expressions, if any.

Target variables in an assertion must occur as <expression1> or as the target parameters of the function in <expression2>. Moreover, the relational operator must be " = ". Examples of assertions are:

```
ASRT:  A > B*SIN(30)  SOURCE:A,B;
ASRT:  A = B*SIN(30)  TARGET:A
                      SOURCE:B;
```

The first assertion tests for the inequality and evaluates to true or false; the second assertion, on the other hand, defines variable A and always evaluates to true.

In addition to arithmatic operators, the + - operator is used in an assertion:

```
ASRT: e1 = e2 + - e3;
```

where e1, e2, and e3 are expressions. The assertion is an abbreviation for the following relationship:

e2 - e3 < = e1 < = e2 + e3

and evaluates to true provided the above relations are satisfied.

Assertions may also be used to specify a relation that must be satisfied by a target parameter of a function in a conjunction. For example, an assertion written as:

```
        CONJ: <J1,J2> = VOLTMETER (<V1)
                        SOURCE:V1;
```

specifies that the value of the target parameter of the function VOLTMETER must be less than V1.

If-clause can be used with assertions just as in if-conjunctions. Syntax of if assertion is:

```
        ASRT: IF <BOOLEAN CONDITION> THEN <ASSERTION>
                                     ELSE <ASSERTION>
                                   SOURCE:<LIST OF VARS>
                                   TARGET:<LIST OF VARS>;
```

The keywords THEN and ELSE may be followed by another assertion which may again have an if-clause. This allows the if-assertion to be nested to indefinite depth. (In the present implementation, the assertion following THEN cannot have an if clause. Thus only a right recursive tree is permitted.)

The if-assertion is taken to evaluate to the same boolean value as the selected assertion following THEN and ELSE. In other words, if the boolean condition in an if-assertion evaluates to true then the assertion is said to evaluate to the same value as the assertion following the keyword THEN, and if the boolean condition evaluates to false, then the assertion is said to evaluate to the same value as the assertion following the keyword ELSE. If an if-assertion (or if-conjunction) defines some variables in its then-part, it must also define exactly the same variables in its else-part.

The concept of free-subscript is introduced next. Its use allows entire arrays to be defined by means of one conjunction or assertion. It also allows relations to be specified between arrays. The notion and use of free subscripts is similar to that in mathematics. For example, the assertion with free-subscript I:

```
ASRT : IF I=1 THEN  F(I) = 1
                ELSE  F(I) = I*F(I-1)
          TARGET:F(I)
          SOURCE:F(I-1);
```

defines the values of F(I) for all values of free subscript I. In other words, it defines the entire array F. Similarly, the assertion with free-subscript I

```
ASRT: A(I) = B(I)
        SOURCE: A(I),B(I);
```

specifies relation between two array variables A and B. This assertion is taken to evaluate to true if the relation holds for all values of the subscript I.

Syntax for declaration of a free subscript is similar to that of an assertion. Statement containing the keyword SUBSCRIPT in the following example:

```
ASRT: I = SUBSCRIPT ('A,B:2',10)   TARGET: I;
```

is used to declare a free subscript "I" for the first dimension of array variable A, and the second dimension of array variable B. The size of the respective dimensions of the variables is ten. Even though the declaration looks like an assertion it should not be confused with an assertion. It declares a subscript which takes values from 1 to 10. The list of variables and their dimensions, i.e. "A,B:2", is called *parent list*.

Subscripts are a powerful way to define arrays. However, certain restrictions have been placed on their use so that the specification may be analysed and an efficient program generated. Let I be a free subscript. A subscript must be in one of the following forms:

1. a subscript term, e.g. I in A(I);
2. an expression of the form (I-K), where K is a positive integer; and
3. another variable or subscripted variable e.g. B(I) in A(B(I)), X in A(X).

For variables which are targets in a conjunction or in an assertion, only the first of the above three forms is permitted.

In the declaration of a subscript the upperbound may be omitted, if it is not known, and replaced by "*". For example, in the assertion:

```
I = SUBS ('F',*) TARGET:I;
```

upper bound of a variable F is unknown. For such variables, the program generator tries to optimize memory. In particular, the program generator allocates memory for 2 elements: current and the previous. Elements corresponding to only the current (i.e. I) and the previous (i.e. I-1) value of subscript may be referenced.

The size of an array variables with subscript I, whose upper bound has been declared to be indefinite, is specified by means of a special array called END-I. Such special arrays are called *end arrays*. The meaning of end-arrays is introduced by means of an example below:

```
I = SUBS ('F',*)  TARGET:I;

IF  I=1  THEN  F(I) = 1
         ELSE  F(I) = I*(F-1)
      TARGET:F(I)
      SOURCE:F(I-1);

IF  I=5  THEN  END_I(I) = TRUE
         ELSE  END_I(I) = FALSE
      TARGET:END_I(I);
```

First statement, in the example above, is declaration for a subscript I. It is followed by an assertion which defines an array F in terms of itself. The second assertion defines an end_array END_I whose first four elements are false and the fifth element is true. This specifies that the size of array F is equal to five. In other words, the size of F is specified to be equal to N such that for index N the value of END_I is true and for all indices less than N the value of END_I is false.

More generally, if "I" is a free subscript then END_I is a multi-dimensional array, its dimensionality being equal to the maximum dimension in the parent list in the declaration of "I". END_I defines the size of those dimensions of the array variables which are in the parent list. For example,

```
          I = SUBS ('G:1,F',*)  TARGET:I;

          J = SUBS ('G:2',*)    TARGET:J;
```

the dimensionality of END_I is one and that of END_J is two. END_I defines the size of the one dimensional array F and the first dimension of two dimensional array G Similarly, END_J defines the size of the second dimension of array G.

Use of free subscripts allows an array to be defined by means of a single assertion or single conjunction. It is important, however, for the variables (be they arrays or scaler) to be single valued. Consequently, a conjunction or assertion which defines multiple values for arrays is invalid. For example, the assertion:

```
           ASRT: A(I) = B(I,J)  TARGET:A(I)
                                SOURCE:B(I,J);
```

is invalid because *it defines an element of array A to be equal to an entire row* (second dimension) of array B. In general, whenever the set of free subscripts associated with a target variable is a subset of the number of free subscripts of the source variables, it defines multiple values for the target variable. There are two exceptions to the above rule:

1. when a source variable containing an extra free subscript occurs as an argument of a reduction function, and the extra free subscript is reduced; or

2. when a boolean condition precedes the assertion. A warning is issued in this case and it is the responsibility of the user to ensure that the target variable is single valued.

Example of an assertion containing a reduction function is:

```
ASRT : F(I) = SUM(G(I,J),J)
     TARGET:F(I)
     SOURCE:G(I,J);
```

Reduction function SUM takes two dimensional array G and sums the elements of the same I index value, thus producing a one dimensional array F. The array variable F is single valued even though the source variable G has an extra subscript J. Example of the second exception is:

```
ASRT: IF END_I(I) THEN  OUT = F(I)
                    TARGET:OUT
                    SOURCE:F(I),END_I(I);
```

OUT is defined by the last element of the array F. However, it is the responsibility of the user to make sure that OUT is not defined by more than one element of F. Nopal program generator does no further analysis to check that it is indeed so.

The Logic component of a test specifies the selection of diagnoses. The diagnoses are selected depending on whether the test evaluates to true or false. The test evaluates to true if all the assertions in the test evaluate to true, and false otherwise. The operators given in Table 4.1 may be specified with each of the diagnoses for their selection.

The logic component is specified by a list, each of whose elements consists of an operator followed by a diagnosis name. For example:

```
LOGIC : *D1, |D2, ~D5;
```

4.3.3 DIAGNOSES

The diagnoses are used to report the result of the test, to isolate failure modes or to elicit a response from the user. It has five parts which can be specified in any order.

1. List of affected components and their failures modes which are isolated by this diagnosis. They may be in conjunctive or disjunctive form where the former means that all the components in the list have failed, while the latter that at least one has failed.

Table 4-1: LOGIC OPERATORS IN A TEST

OPERATOR	MEANING
*	Select the diagnoses unconditionally i.e. irrespective of the outcome of the test.
|	Select the diagnoses if the test evaluates to true.
|~	Select the diagnoses if the test evaluates to false.
&	Mark the diagnoses as selected if the test evaluates to true. The diagnoses should be executed only if all other tests which use this diagnoses (with operators: & or &~) also mark it as selected.
&~	Mark the diagnoses as selected if the test evaluates to false. The diagnoses is executed only if all other tests which use this diagnoses (with operators: & or &~) also mark it as selected.

2. Name of the message to be printed. The message itself is specified separately.

3. Parameters: This specifies the variables whose values must be substituted in the message at the appropriate places.

4. Operator response: It specifies the response from the operator when the generated program is executed. The program waits for a response. Response can be of three types:

   a. press PROCEED key;

   b. press Y(yes) or N(no); or

   c. enter a number.

Pressing the PROCEED key simply causes the program to continue execution. It is typically used to turn knobs and set switches manually, i.e. those which cannot be controlled by ATE. Y or N response is typically used for asking the operator to make a binary choice. The response (c) is usually used to enter reading of meters etc. manually, i.e. those which cannot be taken by ATE.

5. Time: Specifies the real time which must elapse from the start of a test, before the message is issued.

Except for the name of the message and the parameters it takes, if any, all the other parts of the diagnosis are optional.

A diagnosis specification is illustrated below:

```
DIAG D1:
  AFFECTED COMPONENTS = OPEN(RESISTOR1)|OPEN(RESISTOR2),
  PRINT = MSG1,
  PARAMETERS = V,
  TIME = 0;
```

It specifies that at least one of two resistors - RESISTOR1 or RESISTOR2 - has failed due to open circuit, and that message MSG1 with parameter V must be printed. Time 0 specifies that no time delay is necessary in issuing the message.

4.3.4 MESSAGE SPECIFICATION

This specification consists of the text of a message, and parameters and affected components, if any. It implies the printing of the message including the affected components and parameters.

For example, the message MSG1 of Section 4.3.3 can be specified as follows:

```
        MESSAGE MSG1:

          ' ONE OF THE FOLLOWING FAILURES HAS OCCURED: (C). THE
            MEASURED VOLTAGE IS (P).' ;
```

When the above message is printed "(C)" is substituted by "OPEN (RESISTOR1)|OPEN(RESISTOR2)" and "(P)" is substituted by the value of variable V.

4.4 UUTSPECIFICATION

Information relating to the UUT is specified in this section. This allows various consistency checks to be performed within the module. It is organized in two parts: (1) interconnecting points, which are used for identification of the connection points of the UUT to the ATE. (2) component failures which identify all possible faulty components with the failure modes (i.e. types of failures).

A UUT connection point defines a symbolic name for a connection point on UUT, the type of connector used, and the maximum and minimum value of the stimuli which may be applied on it. For example:

```
        UUPT 40 : J1,  CONNECTOR=(A), LIMIT=(VOLT,70,0,GND);
```

J1 is the name by which this connection point is referred to, its type of connector is A, and the maximum and minimum value of stimuli that may be applied with respect to the ground (GND) is 70 volts and 0 volts respectively.

In UUT component failure section, all possible faulty components and their failure modes are listed. Each component specification includes the failure mode, likelihood of the failure, and protection. Protection consists of a list of other components whose failure prohibit testing of this component. For example:

```
COMPONENT FAILURE 2:
  RESISTOR1, FAIL=OPEN, INDEX=1, PROT=(1,11);
```

specifies that the component RESISTOR1 has a failure mode called OPEN, the frequency of failure is 1 (the lower the number the larger the likelihood), and that should the components 1 or 11 fail, tests for failure of this component must not be conducted.

4.5 ATESPECIFICATION

Information relating to the Automatic Test Equipment and the functions used in a module (computational, stimuli or measurement) is stated here. It has two parts: (1) ATE connection point specification, and (2) functions. ATE connection point specification consists of the names of the ATE connections points. Optionally, the specification includes identifying ATE points of the respective UUT points. In the example below:

```
ATEPOINT 1:  ATEPT#30, UUPTS=(J1,J2);
```

"ATEPT # 30" may be connected to UUT points J1,J2. The checking for the UUT points has not been implemented. It is used purely as a documentation device.

Functions used for (1) stimuli, (2) measurement or (3) computational purposes, and (4) for denoting failures are declared in the ATE function specification. Functions in the first three categories are assumed to be defined either by means of other modules or by means of a library of functions. The failure functions (category (4)) are for the purpose of denoting kinds of failure. They are not functions in the sense of the earlier three categories.

Function specification has an item called TYPE which specifies which of the above 4 types does the function belong to, and items PARM and VALUE RETURNED to specify the data types of the parameters and the value to be returned. The number of pins used may also be specified if the function is of type stimuli or measurement.

For example a function PUSH

```
FUNCTION PUSH, TYPE=E, PARM=(INSTACK,S:STACK),
        PARM=(ELEM,S,INTEGER),VALUE RETURNED=(STACK);
```

is of type evaluation. It has two parameters. both source. with the data types STACK and INTEGER. and it returns a value of type STACK. The names INSTACK and ELEM have no significance for the specification. Their use is only for providing mnemonics.

Chapter Five

THE NOPAL PROGRAM GENERATOR

5.1 OVERVIEW OF THE PROGRAM GENERATOR

The Nopal program generator is designed to automate the program design, coding and debugging phases of program development based on a specification in the Nopal language. The program generator analyzes a Nopal module specification, issues a number of reports for the user and, if the module is error free, generates a program in the Equate-Atlas test programming language.

There are three phases in the program generation process. Phase 1 consists of syntax analysis and construction of internal data structures. Phase 2 consists of analysis of the specification for completeness, consistency and non-ambiguity; and of sequencing. In phase 3 Equate-Atlas code is generated. A number of user reports are issued by each of the phases. The three phases are described individually in Sections 5.2, 5.3 and 5.4. More detailed documentation is provided in [46].

The Nopal processor has evolved through numerous revisions over the past several years. The research reported here includes extending the original system [7] with the following capabilities:

1. Modules to provide modularity and abstract data type definition facility.

2. Recursive assertions to allow the arrays to be defined recursively.

3. Declaration of data types and data structures as well as virtual subscripts.

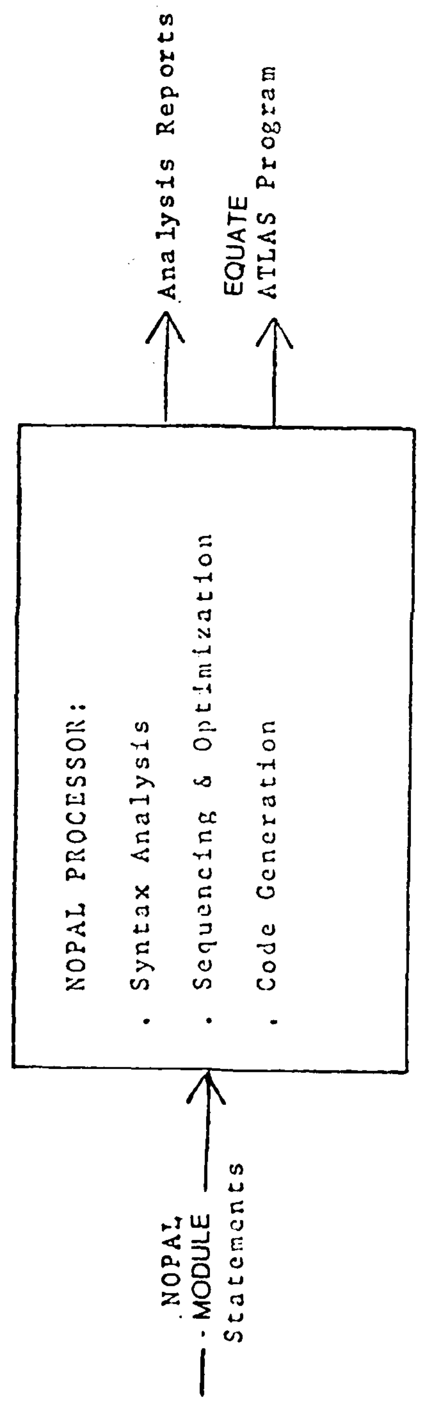


Figure 5-1: OVERVIEW OF NOPAL PROCESSOR

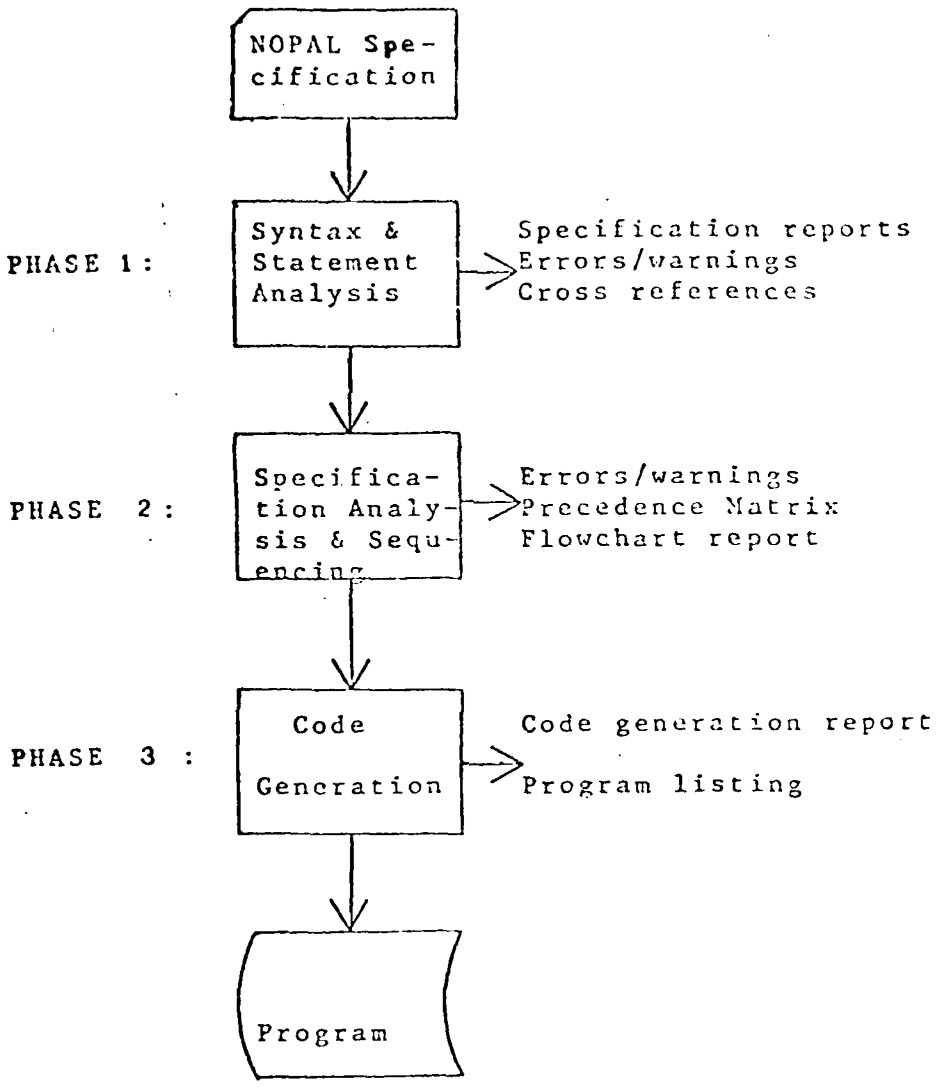


Figure 5-2: MAJOR PHASES OF NOPAL PROCESSOR

4. Input output from secondary storage.

5. Virtual subscripts for efficient utilization of memory.

These extensions entailed modifying some parts of the original system and completely rewriting other parts. In particular, the scheduling algorithm was completely rewritten to handle recursive assertions and input output from secondary storage. It is described in Section 5.3.2.

The code generator was not implemented in the original system [7] but completed later [55]. The Nopal system was demonstrated on an actual Equate Atlas machine [15].

5.2 SYNTAX ANALYSIS AND THE ASSOCIATIVE MEMORY

5.2.1 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 a simulated *associative memory* for ease of further processing. The first phase includes a Syntax Analysis Program (SAP). SAP itself is generated automatically by a meta-processor called Syntax Analysis Program Generator (SAPG), by inputting the formal specification of the Nopal language in a meta language, called Extended Backus Normal Form with Subroutine Calls (EBNF/WSC). SAPG and SAP are described in Section 5.2.2.

SAP incorporates six types of supporting routines, which are composed manually: Lexical Analyzer, Error Stacking, Recognizer, Encoding/Saving/Storing, Semantic Checking and Service Routines. These are described in Section 5.2.2.

At the end of each Nopal statement, a storing routine is invoked to store the statement in the simulated associative memeory using the Store/Retrieve package. The Store/Retrieve package and the associative memory is described in 5.2.4.

Finally, the set of reports generated by this phase are described in 5.2.5.

5.2.2 SYNTAX ANALYSIS PROGRAM - SAP

SAP is generated by the Syntax Analysis Program Generator (SAPG). The input to SAPG is the specification of the Nopal language in the meta-language EBNF/WSC. SAPG and EBNF/WSC were originally developed at the University of Pennsylvania Data Definition Language Project [43] [44]. A brief review of EBNF/WSC and SAPG is given below.

EBNF/WSC extends the standard EBNF to provide semantics. The semantics is specified by means of subroutine names which are included in the productions, along with the terminals and non terminals. These subroutine names indicate the need to call the respective subroutine upon successful recognition of the preceding syntactic unit by the parser. For example, the production

$\langle A \rangle \rightarrow \langle B \rangle$ /aa/ $\langle C \rangle$

indicates that a subroutine named "aa" needs to be called on successful recognition of the non-terminal <B> in the process of recognizing the non-terminal <A>. The subroutines themselves are written manually.

SAPG accepts the specification in EBNF/WSC and generates a recursive descent parser to recognize the syntax defined by the EBNF. Calls are inserted to the subroutines on recognition of non-terminals as specified by EBNF/WSC specification. SAPG requires that the EBNF grammer must be in LL(1) form. The grammer should be free of left recursion, and the first terminal symbol should distinguish between the optional groups at any point in the grammer.

5.2.3 SUPPORTING SUBROUTINES

The EBNF/WSC specification contains names of subroutines which are called from SAP. They can be categorized into six types:

1. *Lexical analyzer* scans the Nopal input string and returns tokens of syntactic units to SAP or the recognizer routines.

2. *Error message stacking routines* help compose and stack error messages before every syntactic unit in the specification. In case of incorrect or missing syntactic units, SAP generates the error message from the error stack.

3. *Recognizer routines* recognize a class of input tokens, such as names and integers. These occur in productions of the form:

   $\langle A \rangle \rightarrow$ /RR/

   where RR is the name of a recognizer routine.

4. *Encoding/saving/storing routines* save the tokens in appropriate data structures and finally in associative memory for purposes of later analysis.

5. *Semantic checking routines* check the local semantics of a Nopal statement.

6. *Service routines* are used by SAP to perform some internal services e.g. popping the error stack.

At the end of each Nopal statement a storing routine is called by SAP, which in turn calls STORE to enter the information relating to the present statement into the associative memory.

5.2.4 ASSOCIATIVE MEMORY AND STORE/RETRIEVE SUB-SYSTEM

The Store/Retrieve Subsystem is a generalized means of storing the Nopal source statements and later retrieving them. It consists of two types of routines:

1. STORE for storing the source language strings, including tokens and entities, gathered during the syntax analysis, and

2. RETRIEVE for retrieving the source strings, and for accessing the "directory entries", the former is through RETREVS and the latter through RETREVD.

The STORE routine is called to create or add to the associative memory, and RETRIEVE to

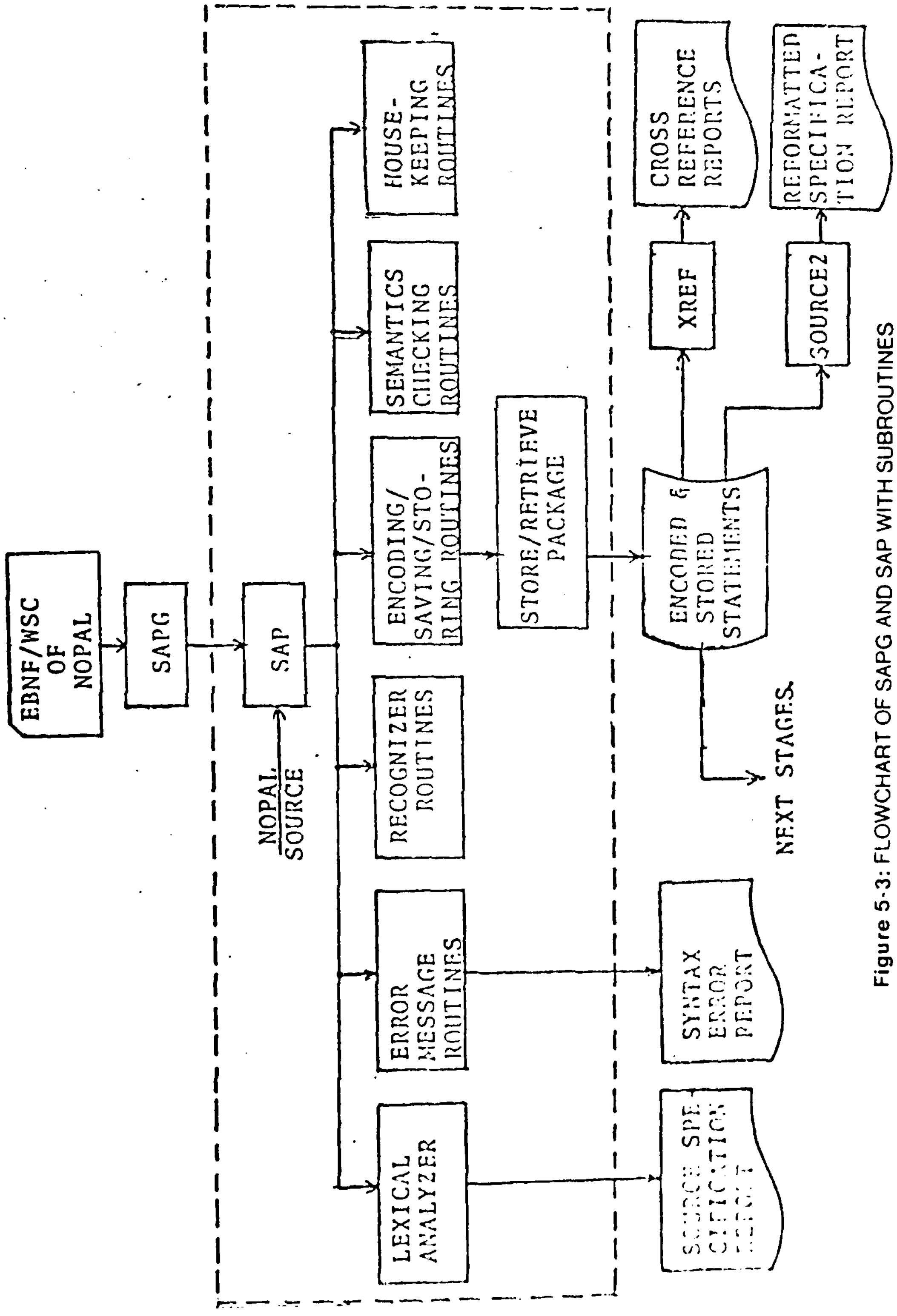


Figure 5-3: FLOWCHART OF SAPG AND SAP WITH SUBROUTINES

access or modify it.

There are eighteen classes of statements and names in Nopal. A list of classes, their mnemonics, and the entities they represent is given in Table 5.1. For example, class 15 represents the variables, and class 2 represents the tests. An identifier occurring in two different classes denotes two different entities. In other words, a name together with its class uniquely identifies an entity.

A directory is used to store all the names and their classes. It is organized as a binary tree according to the lexicographic order of entries. Each node of the tree corresponds to a name and its class. There are cross links in this tree which connect all nodes with the same name together, and all nodes of the same class together. Each node has an additional link (called REFLIST in Figure 5.4) to a storage entry containing this name and type.

There is a *storage entry* for each Nopal source statement. It contains the names (KEY) of all the symbols (rather pointers to the names in the directory) which occur in a the corresponding statement. With each symbol name it has a pointer (called NEXT) which points to another storage entry which uses it. Thus, it provides a very efficient means to find occurrences of symbols in different statements. Associated with each storage entry there is also a pointer (DATA) which points to the entire parsed source string, stored in a separate data area.

The storage entries together with the directory and the data area is called the *associative memory*.

As mentioned in the previous section, STORE is called at the end of each Nopal statement. Its arguments are (1) a list of names and their classes encountered in the statement, and (2) pointer to the data area containing the parsed source statement. It enters the names and their

Table 5-1: CLASSES OF NAMES AND THEIR TYPES

CLASSES	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

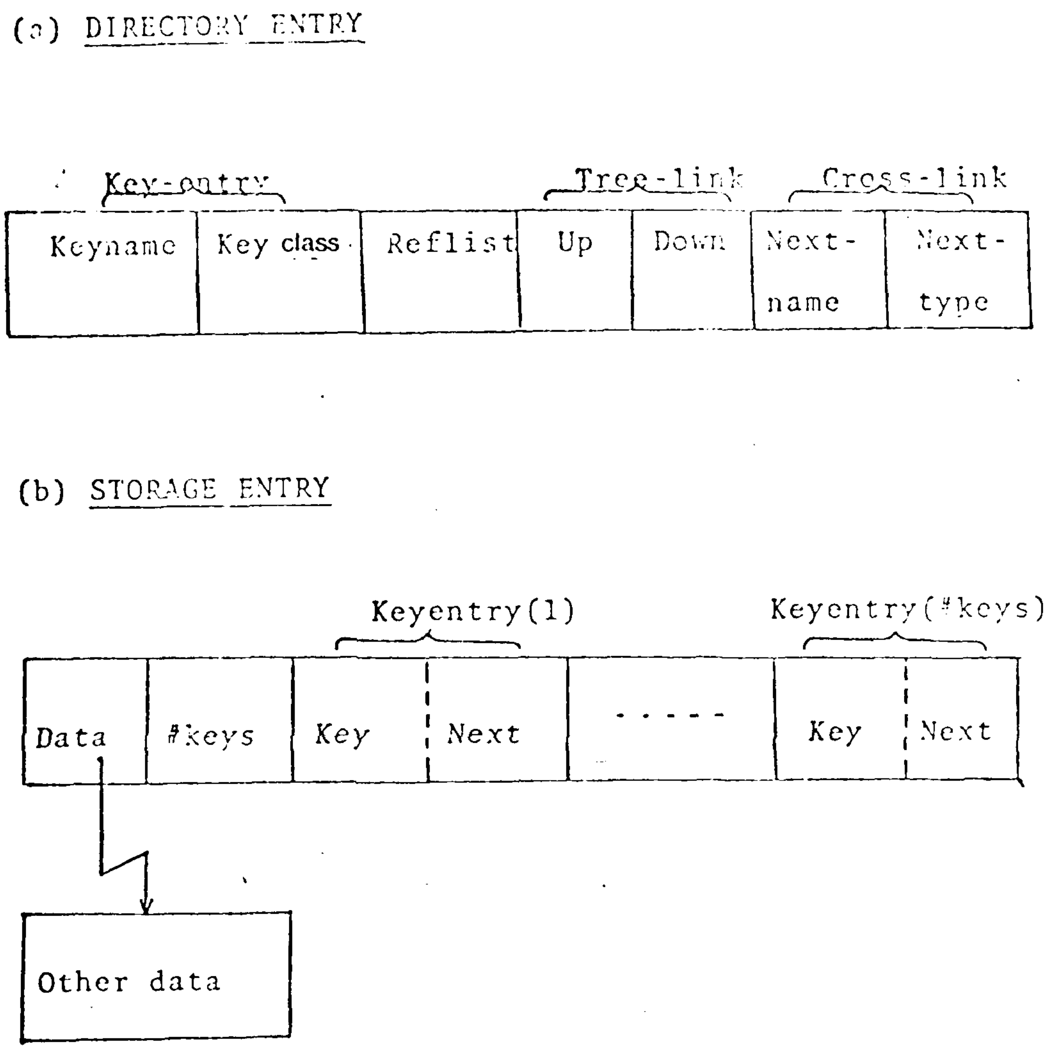


Figure 5-4: STRUCTURE OF THE DIRECTORY AND STORAGE ENTRIES

classes into the directory, and creates a storage entry corresponding to the statement. The storage entry contains the names (rather pointers to the names in directory) which occur in this statement. STORE then proceeds to create all the association links corresponding to each of the names, and also to store the pointer to the data area in the storage entry.

The two procedures RETREVD and RETREVS allow the information to be retrieved from the associative memory. The former retrieves pointers to directory entries, and the latter retrieves storage entries. Entries can be retrieved by logical expressions of their names and classes. For example, all entries belonging to a certain class which do not occur in some statement class can be retrieved by constructing an appropriate logical expression.

5.2.5 REPORTS

Listings of the specification and errors *encountered, if any, in the specifications* are reported at the end of the Syntax Analysis Phase. The programs XREF1 and XREF2 generate a cross-reference listing, and the program SOURCE2 generates a formatted listing of the user specification. Samples of these reports are shown in examples, in the Appendix.

XREF1 also determines the scopes of variables i.e. whether the variables are global or local, and enters it in the data areas of the associative memory.

5.3 SPECIFICATION ANALYSIS AND SEQUENCE DETERMINATION

5.3.1 OVERVIEW

Phase 2 of the Nopal processor analyzes a Nopal specification and determines the sequence of execution of the statements. The analysis is based on a graph representation of the specification. This section presents the background and terminology used in this phase.

Phase 2 is divided into two sub phases. In sub-phase 1 each of the tests is analyzed, and in sub phase 2 the relations between tests within a modfun are analyzed. The sub phases are called *intra test analysis* and *inter test analysis* respectively. In the intra test analysis a graph is constructed for each of the tests. Nodes of the graph represent variables, assertions, conjunctions and diagnoses; and the edges represent precedence relationships between them. In the inter-test analysis, on the other hand, the nodes of the graph represent tests, diagnoses and global variables, and the edges represent precedence relationships between them. Edges in both the sub-phases are labelled to denote the different types of precedence relationships.

The Nopal processor stores the graph in a matrix form. The rows and coloumns represent the nodes, and the entries in the matrix represent edges. A non zero entry, say n, in the position (i,j) in the matrix represents an edge of type n from node i to node j, while a zero entry denotes the absence of an edge.

After the graph is constructed, it is checked for consistency and completeness. It is then checked for cycles, and an attempt is made to eliminate them. Finally, if successful, the nodes are ordered in an execution sequence. Construction of the graphs, consistency and completeness analysis, cycle elimination, and sequencing are described for the intra test sub phase in Section 5.3.2 and for the inter test sub phase in Section 5.3.3. Figure 5.5 shows a flowchart of the processes involved in graph analysis and sequencing.

5.3.2 INTRA-TEST ANALYSIS AND SEQUENCING

Each of the tests in a specification is analyzed in this sub phase. To perform analysis and sequencing, a graph is constructed for each of the tests. Nodes of a graph are: conjunctions,

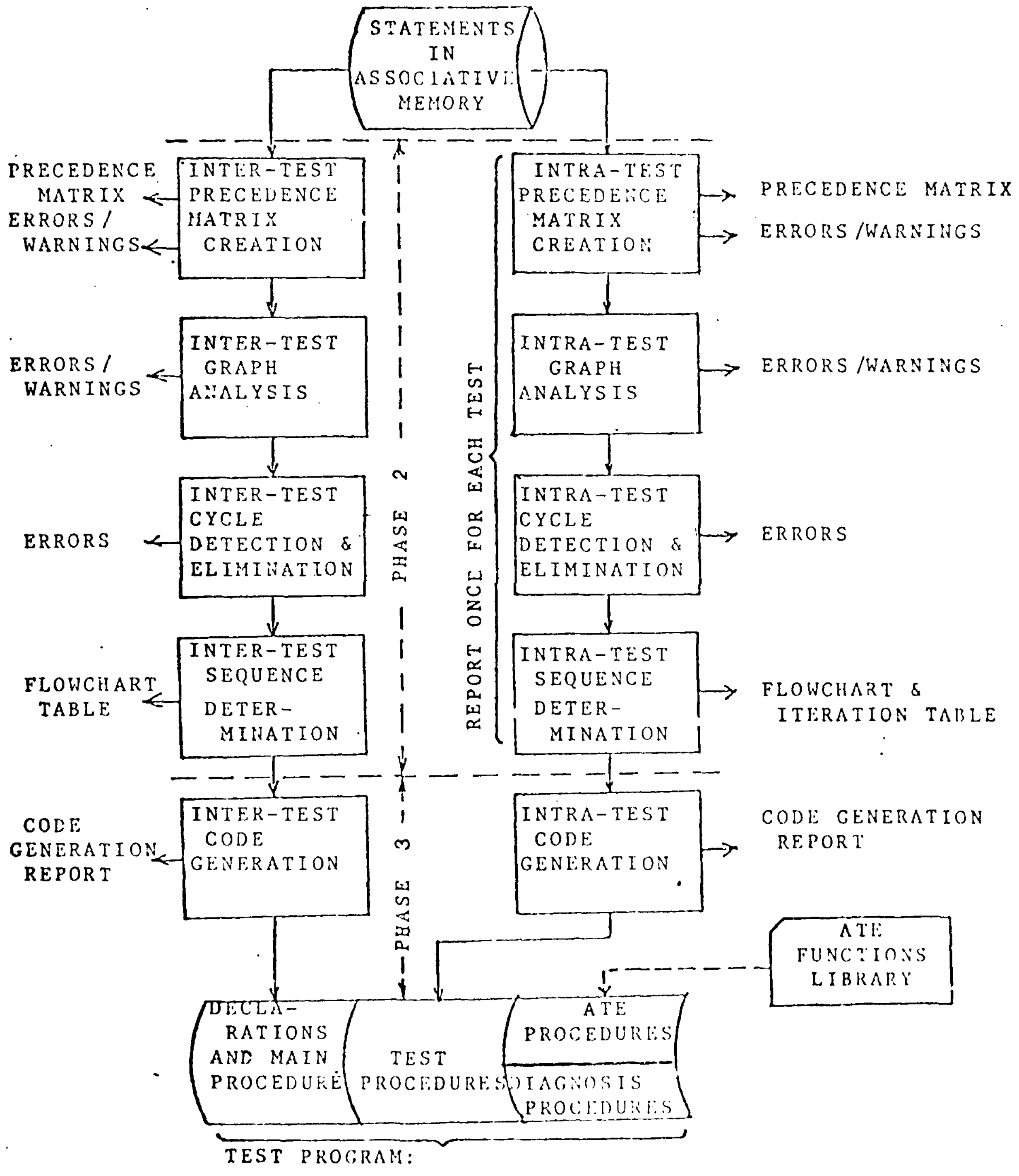

STATEMENTS IN ASSOCIATIVE MEMORY
PRECEDENCE MATRIX
ERRORS / WARNINGS
INTER-TEST PRECEDENCE MATRIX CREATION
INTRA-TEST PRECEDENCE MATRIX CREATION
PRECEDENCE MATRIX
ERRORS /WARNINGS
ERRORS / WARNINGS
INTER-TEST GRAPH ANALYSIS
INTRA-TEST GRAPH ANALYSIS
ERRORS /WARNINGS
ERRORS
INTER-TEST CYCLE DETECTION & ELIMINATION
INTRA-TEST CYCLE DETECTION & ELIMINATION
ERRORS
FLOWCHART TABLE
INTER-TEST SEQUENCE DETER-MINATION
INTRA-TEST SEQUENCE DETER-MINATION
FLOWCHART & ITERATION TABLE
PHASE 2
REPORT ONCE FOR EACH TEST
CODE GENERATION REPORT
INTER-TEST CODE GENERATION
PHASE 3
INTRA-TEST CODE GENERATION
CODE GENERATION REPORT
ATE FUNCTIONS LIBRARY
DECLA-RATIONS AND MAIN PROCEDURE
TEST PROCEDURES
ATE PROCEDURES
DIAGNOSIS PROCEDURES
TEST PROGRAM:

Figure 5-5: FLOWCHART FOR PHASES 2 AND 3 OF NOPAL PROCESSOR

assertions, variables and their ancestors, and diagnoses associated with the test. There are six types of precedence relationships between nodes, which are represented by edges in the graph. Table 5-2 shows the edge types and the relationships that they represent. A priority is associated with each edge; 1 denotes the highest priority and 6 denotes the lowest. Edges with priority 1 are considered mandatory and cannot be deleted. Edges with lower priority are not essential and can be deleted during the cycle elimination stage; they represent preferred relationships rather than mandatory ones.

Relationship of *data determinacy* exists between variables on one hand and conjunctions, assertions and dignoses on the other. A variable node is the predecessor of conjunctions and assertions if it occurs as a source, and is successor if it occurs as a target in them. Similarly, a variable is predecessor of a diagnosis if it occurs as a parameter, and is successor if it occurs in the operator response. *The relationship of data determinacy expresses the idea* that a variable must be defined before it is referenced.

Relationship of *waveform setup* exists between the stimulus conjunction and measurement conjunction. It is not mandatory and is of priority 2. It expresses the notion that stimulus is usually applied before the measurements are made.

Relationship of *diagnosis waveform* exists between diagnosis on one hand and conjunctions and assertions on the other. This relationship is entered as type 16 or 17. A diagnosis which is selected unconditionally on the outcome of a test precedes each of the conjunctions and assertions in the test by an edge of type 16. A diagnosis which, on the other hand, is selected by the logic |, |~, & and &~ succeeds each of the conjunctions and assertions by an edge of type 17.

All the unconditionally selected diagnoses precede all other diagnoses by edges of type

TABLE 5.2 INTRA-TEST PRECEDENCE RELATIONSHIPS

EDGES			SELECTION RULE	
TYPE	PRIORITY	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 5.2 (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

Table 5-2: INTRA-TEST PRECEDENCE RELATIONSHIPS

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

18.

Relationship between nodes in a data structure is called *hieraichichal*, and is entered as edge of type 19. A node in a structure precedes each of its direct descendents in an input structure, and succeeds each of its direct descendents in an output structure.

Pointer relationship exists between a variable which is a key of an ISAM file and the record node of the ISAM file. It is entered as type 20 edge.

Certain data determinacy relations are identified as *recursive* and are entered as type 21 edges. They are mentioned here for the sake of completeness and are described later after a discussion of array graph.

An array is represented by a single node in the graph: An array variable is represented as a single node independent of its dimensions, similarly an assertion is represented by a single node irrespective of the free subscripts which occur in the assertion. Edges in the graph represent relations between the nodes. An edge between two nodes, when at least one of the nodes represents an array variable, denotes an array of relations (*array relations*) between the two nodes. A graph whose nodes represent array variables and whose edges represent array relations is called an *array graph*.

The intuitive notion of array graph, introduced above, can be made more precise in terms of, what is called, the *underlying graph* (UG) of a specification. A variable node of a UG represents a simple variable, conjunction, assertion, or diagnosis. In other words if B is an array variable of two dimensions of size 5 and 10 each respectively, then a separate node is needed in the UG to represent each of the 50 elements of B similarly, for conjunctions, assertions and diagnoses which have free subscripts and express array of relations between array variables, a node represents an element of the array of relations, e.g. if B is an array as before and it occurs in a conjunction, assertion or diagnosis, then there will be 50 nodes, each representing a conjunction, assertion or diagnosis involving one element of B. Edges in a UG represent relationship between nodes as outlined before, and as the nodes represent simple entities the edges also represent simple relations.

Array graph (AG) can be formed by taking the union of nodes and edges in an underlying graph as follows: represent the nodes, say N_i Vi, in UG representing different elements of an array variable (or conjunction, assertion or diagnosis) by a single node, say N, in the corresponding array graph; and for an edge from any of the nodes N_i to any other node, say P_i, in UG form an edge from N to P in AG. The resulting smaller graph is an AG of the given UG. Thus, AG is a compact way to represent UG. The UG may be an enormous graph which is *impractical* to analyze.

An array graph is shown in Figure 5.6. It is for the familiar PUSH function of stack data type.Nodes S.Z, $\alpha 2$ and S1.Z are array nodes in the example. The rest of the nodes represent simple variables and assertions.

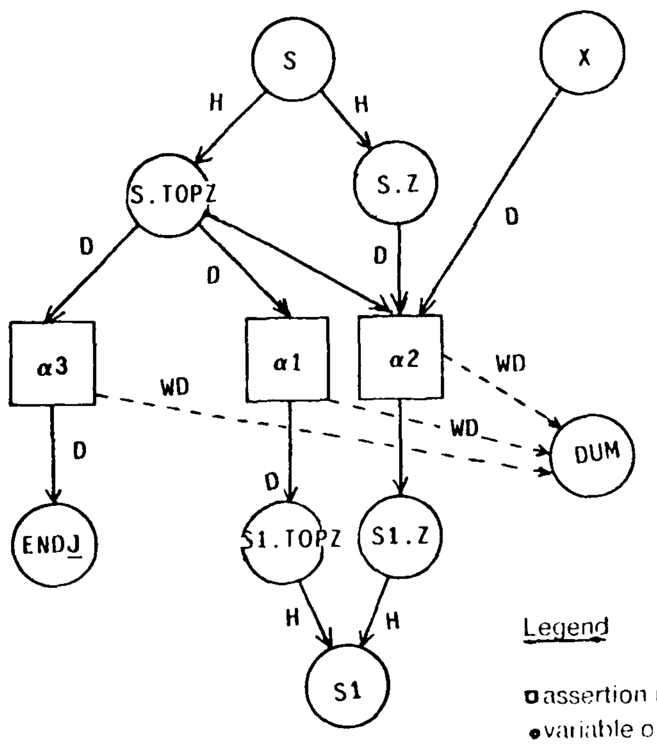
The array graph is constructed by means of the procedure INTSEQ. It is first analyzed for two types of consistency checks.

1. Single assignment rule: Variable nodes should have exactly one predecessor assertion or diagnosis, i.e. a variable should be defined by exactly one assertion or diagnosis etc. (In case the variable is part of an input structure or is a source parameter of the modfun, then it need not have any predecessor in the graph. In the first case, the input file is taken to be the implicit predecessor. In the second case, the value of the formal parameter is defined when the modfun is called and hence it does not have a predecessor in the present graph.)

2. Target variables in an assertion should occur either on the left hand side of the relational operator, or as target parameters of function. Target variables of a conjunction should occur only as parameters of functions.

```
1         MODAL MODULE STACK;
2             DCL 1 STACK: RECORD,
2               2 TOPZ: INTEGER,
2               2 Z: INTEGER ARRAY(100);

3           MODFUN PUSH(S:S STACK, X:S INTEGER) RETURNS(S1: STACK);
4              STIM;
5                 ASRT: J = SUBS('S.Z, S1.Z,END_J',100)    TARGET:J;
6        α1       ASRT: S1.TOPZ = S.TOPZ + 1    TARGET:S1.TOPZ
6                                 SOURCE: S.TOPZ;
7        α2       ASRT: IF J=S.TOPZ THEN S1.Z(J) = X
7                    ELSE S1.Z(J) = S.Z(J)    TARGET:S1.Z(J)
7                                 SOURCE: S.TOPZ,S.Z(J);
8        α3       ASRT: IF J=S.TOPZ THEN END_J(J) = TRUE
8                                     ELSE END_J(J) = FALSE
8                         TARGET:END_J(J)
8                         SOURCE:S.TOPZ;
9              LOGIC: IDUM;
10          DIAG DUM: PRINT = MSG;
```



Legend

□ assertion node
● variable or diagnosis node
H hierarchichal edge
D data determinacy edge
WD waveform-diagnosis edge

Figure 5-6: SPECIFICATION OF PUSH AND ITS ARRAY GRAPH

5.3.2 INTRA-TEST ANALYSIS AND SEQUENCING

Subscript analysis is performed by the procedure SUBANAL. First, all the declarations

free subscripts are checked for correctness i.e. that no subscript is declared twice and t

the same variable dimension is not used in two subscript declarations. Finally, a table of

the free subscripts is constructed containing their names and upper bounds.

Further checks of the free subscripts are conducted by the procedure SUBUSAG

follows:

1. All occurrences of a target variable in an assertion or conjunction must have the same free subscript. For example, In the assertion (from PUSH example)

```
IF J=S.TOPZ THEN S1.Z(J) = X
             ELSE S1.Z(J) = S.TOPZ
     TARGET:S1.Z(J);
```

   variable S1.Z(J) is the target variable occurring in both the THEN and ELSE parts.

2. The free subscripts of the source variables must appear as subscripts with the target variables. There are, however, two exceptions. First, a free subscript which is reduced by a reduction function, need not appear as a subscript of the target variable. Second, in an if-assertion or if conjunction the free subscript of source variables need not appear with the target variables. In that case, a warning is isued to the user that the target variable should be checked that it does not have multiple definitions. A warning will be issued in the following case:

```
IF END_J(J) = TRUE THEN OUT = A(J)
              TARGET:OUT;
```

3. Subscripts must be in one of the following forms:

   a. a subscript term e.g. I in A(I);

   b. a subscript expression of the form (I-k) where I is a free subscript and k is a positive integer;

   c. another variable e.g. B(I) in A(B(I)) or X in A(X).

4. If a positive integer or a subscript term appears as an index of an array variable V, and the range for the corresponding dimension of V is declared to be d, then in case of the integer its value, and in case of the subscript term its upper bound, should be less than d. In other words, a subscript expression is checked to see that it lies within the range for the above two cases.

The array graph is analyzed, next, for cycles by the CYCLES procedure. If a cycle i detected in the graph then the procedure attempts to eliminate it. Edges in a cycle havin priority value greater than 1 may be removed. These edges correspond to two cases:

1. The edges are considered as preference edges and are not essential e.g. a type 2 edge between stimuli conjunction and measurement conjunction denotes the usual situation that the stimulus is applied before the measurement. However, there may be situations in which a stimuli depends on the measured value, and can only be applied after the measurement is performed.

2. The edges do not imply a cycle in the underlying graph. This occurs with recursive edges. For example, the array graph for an assertion of the form:

```
α: IF I=1 THEN A(I) = 1
           ELSE A(I) = I*A(I-1)
        TARGET:A(I);
```

   is given by Figure 5-7. In the underlying graph recursive edges give rise to acyclic spiral like structures.

In the event that deletion of edges fails to resolve all the cycles, an error is reported to the user that the specification contains a circular definition and that it is not possible to sequence it. A warning is issued for each edge deleted in the cycle elimination process.

Procedure PROPAGT determines the relation between the nodes and the free subscripts in order to find the proper scopes for each possible iteration. This procedure constructs fo each node a list of subscripts on which it depends, and hence the list of iterations in which i should participate. In the Nopal language, iterations result from explicit appearance o subscripted variables and subscripts themselves.

The final stage is to sort the nodes into a possible execution sequence. If there were nc subscripts, and hence no iterations, then a simple topological sort would be sufficient. The presence of subscripts introduces an additional factor. A brute force approach to handle subscripts is to enclose each node in the iteration scopes of its subscripts, with the exception

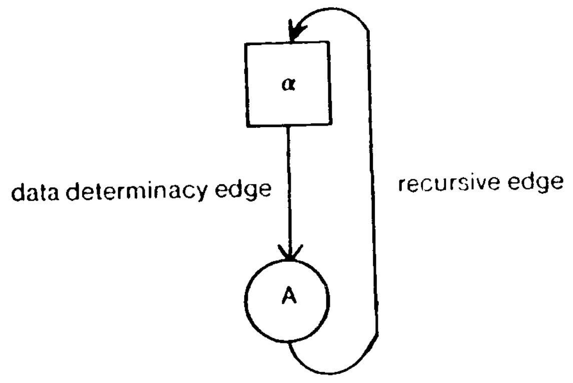


Figure 5-7: RECURSIVE EDGE

that the nodes belonging to a recursive cycle (which was opened during cycle elimi stage) must be enclosed in the scope of a common subscript iteration. The algorithm u the Nopal processor does better than the brute force approach and tries to maximi scopes of iterations. It is performed by the procedure SCHEDLR. The scheduling proc described below. At the end of this process an order vector is generated along with s and subscripts of iteratons.

There are three inputs to the scheduling process:

1. an array graph,

2. a list of free subscripts for each of the nodes and a list of nodes for each free subscript, and

3. a list of recursive cycles, where for each recursive cycle there is a list of nodes which occur in it.

The scheduling process consists of two procedures: SCHEDLR and ORDERER. SCH calls on ORDERER requesting trial ordering of nodes depending on a subscript. Bas the results of the trial ordering, SCHEDLR calls ORDERER a second time to pe actual scheduling of nodes.

SCHEDLR makes use of a stack called NEST, which contains free subscripts. stack contains free subscripts corresponding to which iterations have begur iterations are nested within each other, with the free subscript on top of the NI representing the innermost iteration. Nodes currently being ordered by ORDERER a within all these iterations. In addition to the NEST stack, SCHEDLR also has a s subscripts called TBNEST. It contains those free subscripts which are candidates nested at the innermost level in the iterations corresponding to the free subscripts stack.

SCHEDLR has three phases. In Phase I it picks up all those nodes in the gra have no predecessors, and finds the union of the free subscripts associated with the subscripts form the set CANDLIST. If one of the nodes does not depend on any then it is treated as belonging to subscript free set, and a special entry (zero) is i CANDLIST. From the set CANDLIST, another set called BESTCANDLIST is for following cases are performed progressively in succession until a non empty BESTC results:

1. all the subscripts in set CANDLIST which also belong to TBNEST are place BESTCANDLIST;

2. a subscript in the CANDLIST set which also belongs to NEST stack is place BESTCANDLIST;

3. if entry zero, corresponding to subscript free nodes, belongs to CANDLIST placed in BESTCANDLIST; otherwise

4. all entries belonging to CANDLIST are placed in BESTCANDLIST.

BESTCANDLIST now contains those free subscripts which are possible candidat ordering process.

In Phase 2, SCHEDLR calls ORDERER with each of the entries in BESTC

perform a trial schedule. The results of the trial schedule are evaluated acco

priority table given in Table 5.3.

Let the ORDERER be called to perform a trial ordering with a subscript IC belo

set BESTCANDLIST. It performs a trial schedule and returns the following informa

1. whether all the nodes depending on IC got scheduled, i.e. whether IC *completely scheduled*;

2. a set of other subscripts, IOC, which got *completely scheduled*;

3. a set of other subscripts, IOP, some of whose nodes got schedulec subscripts which got *partially scheduled*; and

4. whether there is a recursive cycle some, but not all, of whose node scheduled.

A priority value is evaluated based on the above results as per Table 5.3. (In the

true if condition (1) is satisfied, and CY is equal to 0 if condition (4) is satisfied.)

If the priority value is 1 then the SCHEDLR proceeds to Phase 3 to call the (

second time with IC, this time to do the actual scheduling. If the priority value is gr

then trial schedule for a new subscript in BESTCANDLIST is done. Finally, w

subscripts in BESTCANDLIST have been trial scheduled, the subscript, say IH, with

value of priority, say P, is chosen. If P is equal to 5 an error message is issued that

cycle needs to be broken. If P is equal to 4 a warning is issued indicating to th

some files must be entirely located in the main memory. For other values of P it p

Phase 3 of SCHEDLR to do the actual scheduling with subscript IH.

Phase 3 corresponds to the second call on ORDERER to do the actual sch

CP ↓	IC has ↓	IOP = NULL
		CY = 0
			IOP has no I/O
			CY = 0
				IOP has I/O
				CY = 0

CP	I/O	1	2	4
	no I/O	1	2	3
~CP	I/O	4	4	4
	no I/O	2	3	3

I/O stands for input-ouput. Meaning of IC, IOP, CP, CY etc. is explained in th

Table 5-3: PRIORITIES OF THE TRIAL SCHEDULE

nodes. Following this, the nodes which got scheduled are removed from the graph. Finally, if some nodes still remain in the graph, Phase 1 is started all over again.

Procedure ORDERER is described next. It has two modes: (1) to perform trial scheduling, and (2) to perform actual scheduling of nodes. It has a parameter, IC, which gives the name of a free subscript.

In mode (1) it does a topological sort of the nodes which depend on the subscript IC. It then evaluates the result of the trial ordering and returns the result in CP, IOC, IOP and CY as described earlier.

In mode (2) it performs a topological sort of the nodes which depend on the subscript IC, on all of the subscripts in the stack NEST and on no other subscripts. The resulting ordered set of nodes is added to the ORDER vector and removed from the graph.

The scope of iterations is determined from the NEST stack. Each time an entry is pushed on the NEST stack it defines the beginning of a new iteration; and each time the NEST stack is popped it defines the termination of an iteration. The NEST stack is updated as follows:

1. Each time the ORDERER is called to do the actual scheduling with a subscript IC, an entry is made on NEST provided IC was not selected by case (2) in Phase 2 of SCHEDLR.

2. The NEST stack is popped (in case(2) in Phase 2 of SCHEDLR) until the selected subscript IC is on top of the NEST stack.

The final result of scheduling process is an order vector and an iteration table giving the scopes of iterations. They are used by the code generation phase (by procedure CDETEST described in Section 5.4) to generate Equate-Atlas code.

5.3.3 INTER-TEST ANALYSIS AND SEQUENCING

In this sub-phase a graph is constructed for the entire modfun specification. The nodes of the graph are tests, diagnoses and global variables. There are seven types of precedence relationships between nodes. They are described in Table 5.4. The table gives the name, type, and priority associated with a precedence relationship. It is followed by a description of the predecessor and successor nodes which satisfy the relationship. The meaning of the terms: type, priority etc. is the same as discussed in Section 5.3.2. The seven relationships are described below.

Data determinacy relationship exists between tests, diagnoses and global variables. A test or a diagnosis is the predecessor of a variable if the variable is defined by one of them, and successor if the variable is used as a source by them.

Interactiveness relationship exists between a diagnosis and the test which selects the diagnosis by "after" and "after not" logic operator (A and A~). It means that the test is selected based on the operator response to the diagnosis.

Component protection relationship exists between a diagnosis and a test, if the test has an affected component which is protected by the diagnosis. Its purpose is to inhibit testing of a component if another component which protects it has failed.

Fault isolation relationship exists between a diagnosis and a test, if the set of affected components of the diagnosis contains the set of affected components of the test. It expresses the idea that those tests which isolate smaller number of failures should be performed later compared to the tests which isolate larger number of failures.

Stimuli-application relationship exists between two tests if one of the tests has stimulus funcions which are applied more frequently than those in the other test. It leads to performing

TABLE 5.4 INTER-TEST-MODULE PRECEDENCE RELATIONSHIPS

Precedence			Relationship selection rule		Run-time	
Type	Prio-rity	relationship	PREDECESSOR	SUCCESSOR	condition	Explanation
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 subset of D's.	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 5-4 (continued)

Precedence			Relationship selection rule		Run-time condition	Explanation
Type	Prio-rity	relationship	PREDECESSOR	SUCCESSOR		
10	4	Failure likeli-hood	Test module whose smallest failure index of affected components is smaller	Test module whose smallest failure index of affected components is larger		Tests whose compo-nents 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 con-cludes. Types 11 through 15 may be combined into a type, but they are separated to speed up later process-ing.
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"(&¬)		

of as many tests as possible once an ATE device is connected.

Two tests are related by means of *failure liklihood relationship* if the predecessor test isolates those failures which are more likely to occur. The liklihood of a failure is supplied by the user in the specification.

Finally, a test is the predecessor of a diagnosis selected by one of the logic operators: |, |~, & and &~ in the test. The selection of a diagnosis by means of one of the above logic operators is dependent on the outcome of the test. This is expressed by the *logic operator relationship*.

Several of the relationships described above are not mandatory. They represent desirable but not necessary relationships; in other words, such relationships are good for efficiency but not necessary for correctness. A priority value greater than 1 is associated with them in Table 5.4.

Procedure EXTSEQ constructs a graph for the modfun specification. The nodes of the graph represent simple entities, unlike the graph for a test. This is so because there are no free subscriptsfree subscript associated with the entities which are represented by the nodes. The graph is analyzed to check:

1. that every variable node has a predecessor, i.e. every variable is defined;
2. that every variable node has only one predecessor, i.e. every variable is defined by only one test or diagnosis;
3. that a diagnosis does not precede two or more tests with type 2 or 3 edges, i.e. a diagnosis does not select more than one test by the logical operators A and A~.

The next step is to detect cycles and, if possible, eliminate them by removing edges with priority value greater than 1. They correspond to preference edges and are not essential e.g.

a failure liklihood edge (type 10 priority 4) between two tests expresses that the liklihood of detecting a failure by the predecessor test is higher. It reflects knowledge which may be useful for quicker fault isolation, but is not mandatory for correct fault isolation.

The final step is to sort the nodes into a possible execution sequence. A simple topological sort is sufficient because there are no free subscripts or iterations. Finally, an order vector is generated. The order vector is used by Phase 3 to generate Equate-Atlas code.

5.4 CODEGENERATION

This is the third and final phase which generates Equate-Atlas code corresponding to the Nopal specification. Code is generated for each of the entities: tests, diagnosis, assertions, conjunctions, variables, structures etc. The order of execution of these entities is determined in Phase 2 in the form of an order vactor. This is used in the present phase in generating the sequential program in Equate-Atlas.

Eqate-Atlas is a test programming language and is a subset of IEEE standard Atlas. It has a number of features to support the programming of ATE. However, it does not support many of the widely accepted programming constructs. Mostly notably,

1. The procedures in Equate-Atlas do not have parameters. The procedure simply represents a body of sequential code which is executed when called.

2. There is no provision for local variables in procedures. All variables are considered global.

3. The if-then-else construct is absent. It can be simulated using "compare" and "goto" statements, reminiscent of the assembly language instructions.

4. It does not have a do-while costruct. It has for-loop similar to the Do-loop in Fortran.

5. It does not have a linking facility. Consequently, all the procedures should be included in one program at compile time. The procedures communicate by means of global variables.

6. The only data structuring method in the language is array. Declarations for records and structures are not allowed.

7. The language has only two data types: decimal and digital. Decimal is used for floating point numbers and digital for bit strings. There are no other data types e.g. character strings, integers etc.

8. It does not allow dynamic memory allocation.

Certain conventions were established, in view of the rather severe limitation given above. For example, to pass parameters to a procedure, named say P, the following convention was adopted: The parameters were passed in the special variable names P.PRMO1, P.PRMO2, ... and so on. At the time of the procedure call these parameter variables are given values. The body of the procedure uses these names to receive source parameters and defines values of the target parameters to return values.

Lack of a linking facility forces that the Equate-Atlas statements generated separately for each of the modules be put together and the resulting total program be complied at one time. This raises a problem, however. The language has no local variables, and hence, two variables of the same name occurring in the two different modules would be treated as one. The clash of the variable names is avoided by generating unique names for the module. All variables in a module are suffixed by "." followed by the module name.

The absence of if-then-else and do-while is handled using the primitives "compare" and "goto". The absence of structure declaration is handled by simple variables and arrays. Although, these make the generated code messy, they pose no conceptual problem.

The code generation phase consists of three sub-phases. Sub-phase 1 consists of generation of program header, declaration for the global variables and system variables, and a procedure definition for each of the diagnosis. Sub-phase 2 consists of generation of a procedure for each test. The third and final sub-phase 3 consists of generation of logic and sequence of calls on procedures for tests within a modfun. The procedures for each of the sub-phases are CDEMAIN, CDETEST and TRMNATE respectively. The highlights of the sub-phases are presented below.

In sub-phase 1 the declarations for global variables are issued. For a simple variable in the Nopal module, a simple variable by the same name (suffixed by the module name as described earlier) is declared in Equate-Atlas. Similarly, for an n-dimensional array variable in Nopal, an array with the same upper bounds for each of the dimensions is declared in Equate-Atlas. There is an exception, however. For array dimensions whose upper bound has been declared as "*" in Nopal, and for which only two elements - current and the previous need to be kept in memory, size of 2 is declared in Equate-Atlas. The two elements in Equate-Atlas are used to store the current and the previous value of the elements of memory and is part of the space optimization done by the Nopal processor.

Equate-Atlas does not have any facility for declaration of structures. Consequently, a translation is made: the fields in the structures are declared as variables. The dimensionality of the variables is the same as the dimensionality of the fields. (The dimensionality of a field is obtained by propagating the dimensionality of its ancestor nodes in the structure, to the field. This is done by the procedure XREF1.)

In case of a module M (not the main module) an additional dimension is added to the fields of the record which gives the representation for the abstract data type M. For example, in the declaration of representation of a stack

```
NOPAL MODULE STACK;
    DCL  1 STACK: RECORD,
           2 TOPZ: INTEGER,
           2 Z: INTEGER ARRAY(100);
```

the fields TOPZ is declared to be a one dimensional array, and Z a two dimensional array. This extra dimension is added to allow storage of all the variables of type stack. Similarly, in the usage of the fields of stack, S.TOPZ, the qualifier S becomes an index which provides the reference into the array TOPZ. All this became necessary because the object language does not have facili[illegible] for dynamic memory allocation. In PL/I for example, the above could have been implemented by means of based variables and pointers. Record STACK with components, i.e. variable TOPZ and one dimensional array Z, would have been declared as a based structure. The qualifiers would simply have been pointers. There would be no need to add an additional dimension.

It follows from the above discussion that, in the current implementation, the variables of abstract data types are indices to arrays in the Equate-Atlas program and store decimal numbers.

Procedure CDETEST which performs sub-phase 2 is called at the end of sequencing of each test in the intra-test analysis and sequencing. CDETEST generates a procedure for the test. The body of the procedure contains sequential code corresponding to the conjunctions, assertions, and logic for selecting diagnoses. Statements for conjunctions and assertions are generated one at a time in the order deteremined from the earlier intra-test sequencing phase. Iteration statements are also generated based on information about the name of the iteration variable, its upper bound and its scope. In cases, where the upper bound is not specified an arbitrarily large upper bound is used. However, the appropriate termination condition is generated to exit the loop.

In the case of input (or output) structure, calls are generated on the ACCESS (or SAVE) functions to read (or write) the structure from (or to) the appropriate file. Similarly, in case of abstract structures of abstract data type, say DT, calls are issued on ACCESS_DT or SAVE_DT as the case may be.

Finally, sub-phase 3 generates calls on the procedures for the test, and the logic which precedes these calls. The order in which these calls are generated depends on the ORDER vector produced by the inter-test analysis and sequencing phase. This sub-phase is not needed for modules (except the main module) because only one test per modfun is permitted in the present implemenation.

At the end of the three sub-phases, Equate-Atlas code is generated for a module specification. This can be put together with the code for other modules, thus yielding a complete Equate-Atlas program. One of the modules must be a main module. This program can now be run on an Equate-Atlas machine.

Chapter Six

CONCLUSIONS AND FUTURE WORK

6.1 SUMMARY

This dissertation presents the approach of abstract data types to introduce modularity in non-procedural languages. It introduces the notion of *module* for the specification of an abstract data type in a non-procedural language based on equational specification. A module specification can be analyzed for consistency, completeness and non-ambiguity independent of other modules. It allows abstract data types to be specified independent of their use. The concept of module is general enough to allow the specification of recursive data types.

A simple equational language is introduced, and the least fix point semantics of modules is presented. It is shown by means of an example how a data type specified by a module satisfies certain algebraic axioms.

Nopal, a non-procedural language designed for automatic tes ing of physical systems is used as an example to show the feasibility of the use of abstract data types. Nopal language allows abstract data types to be specified by means of modules. The data types once specified can be used in other modules.

A complete implementation of the Nopal program generator is described in brief. A number of examples and their sample runs are given in the Appendix. The program generator analyzes the specification and generates an efficient program in Equate-Atlas satisfying the specification. Optimization for memory and execution time is done in the generated program.

The use of abstract data types allows relations to be specified between variables which are not just of elementary type, but are of arbitrary type. It allows a data type to have an arbitrary degree of complexity hidden away in the module and shielded from its use. This is of particular advantage in decomposition of the problem. It allows operations on larger units of data, ignoring lot of detail, in the process. When these larger units of data correspond to some concept naturally occurring in the problem domain (e.g. stacks and tokens while parsing a string in a formal language) the specification is written in terms of these concepts. It also allows devices in automatic testing to be treated as data types.

Procedures or subroutines are procedural abstraction in the conventional programming languages. They represent a form of abstract action which fits well with the prescriptive style of programming. In non-procedural languages, on the other hand, the relationships between variables is the building block. Use of abstract data types allows the variables to be used and their values defined free of the details of the arbitrarily complex data structure that they might represent. It is felt that just as the procedures are a natural way to introduce modularity in procedural language, the abstract data types are a natural way to introduce modularity in non-procedural languages.

A most important feature of the introduction of the abstract data types in the non-procedural language has been that it does not lead to a change in semantics of the non-procedural language. This is in contrast to procedural languages where an abstract data type facility has led to an object oriented semantics e.g. CLU (Section 2.2), which is different from conventional and generally accepted value oriented semantics.

6.2 FUTUREWORK

Work needs to be done in two areas to make the abstract data types easy to use in non-procedural languages:

1. Efficiency of the generated program should be improved upon, and

2. Additional extensions should be made to the language.

Some of the problems are outlined below.

6.2.1 EFFICIENCY CONSIDERATIONS

In the current implementation, Nopal processor does the following memory optimization: If an array variable is declared to be of dimension '*' and is used such that only the current and the previous element of the array is needed, then the memory allocated for the array is equal to two elements. This is of great value when the array is an input/output structure a.d represents a big file on secondary storage. The above should be extended to not just 2 but *k* elements of an array variable (where k is an integer). It should be determined when a constant storage, k, may be used in the generated program automatically without having the user to declare it. This is part of ongoing research by Mr. K.S. Lu [36] to generate efficient programs from a non-procedural specification (and is independent of the use of abstract data types).

Modularity makes some of the optimizations impossible at program generation time. For example, in the generated program for the specifiaton of an abstract data type storage is allocated for the representation of each of the variables of abstract data type. Even when some of the variables are not needed at the same time, memory optimization cannot be done at the program generation time, since the use of the abstract data type is independent from its specification, and the corresponding information is not avalable at the time the program is generated from the specification.

A possible way to reuse the memory which is no longer needed is to do garbage collection. Any one of the well known techniques can be used [31] to provide better utilization of memory.

6.2.2 LANGUAGE EXTENSIONS

The data typing facility described here can be extended in many ways to improve compactness, clarity etc. To give an example: The language should permit (and the implementation should support) the specification of "parameterized" data types. By this it is meant that the specification of the abstract data types contains a data type as a parameter. For instance, it should permit the specification of a stack of type T, where T can be integer, character, stack etc.; and is specified with the use of the generic stack. This would allow a single stack specification to represent the various types of stack and lead to compactness as well as economy of names of data types and their modfuns.

Non-procedural specification should be investigated in the light of distributed processing. The applicative nature of the language is ideally suited for detection of parallelism within a module. The array graph can be directly translated into a parallel program. Research needs to be conducted to allow different modules to execute on different processors and communicate with each other.

Appendix A

EXAMPLES OF NOPAL SPECIFICATIONS

Some example specifications are given here. Sample runs for specification of stack together with a complete set of reports generated by the Nopal processor are given in the first section. The sections which follow, contain other example specifications with sequencing report for each of the specifications.

A.1 STACK

Nopal specification for stack is given in this section. Nopal module STACK defines a representation consisting of TOPZ and Z. It is followed by a specification of the functions PUSH, POP, TOP, and EMPTYSTACK which can operate on stacks. Each function implicitly has a test and is specified by means of assertions. The assertions specify relationships between the input and output parameters of the function. For example, in statement number 6 (which is an assertion in modfun PUSH) value of TOPZ component of stack S1 is specified in terms of the value of TOPZ component of stack S (where S1 is the output parameter, and S the input parameter of PUSH). The stack has been discussed quite extensively in Chapter 3 and the various assertions are not discussed any further here. Since a logic component must be associated with a test in Nopal, a dummy diagnosis is specified. Similarly, since the assertions must be part of stimuli or measurements, the assertions have been arbitrarily placed under stimuli in each of the modfuns.

A sequencing report is generated for each of the functions by the Nopal processor. It consists of weighted adjacency matrix representing the array graph; followed by an order vector which represents an ordering of the nodes of the graph. The order vector determines the sequence in which Equate-atlas code is generated for the module. Equate-Atlas code is given after the sequencing report. It is followed by a cross-reference and attribute report, and warning messages.

```
       /* NOPAL TEST SPECIFICATION SOURCE INPUT, FILE: SAPLIST */

NOPAL PROCESSOR OPTIONS SPECIFIED: SAPLIST,XREF1,CODE,SEQ,NOXREF2,SOURCE2,TRACE=4,DEBUG=SEQ.CODE,LINE=5000


STMT NO.
--------

    1       NOPAL MODULE STACK;
    2          DCL 1 STACK: RECORD,
    2            2 TOPZ: INTEGER,
    2            2 Z: INTEGER ARRAY(100);

    3         MODFUN PUSH(S:S STACK, X:S INTEGER) RETURNS(S1: STACK);
    4            STIM;
    5               ASRT: J = SUBS('S.Z, S1.Z,END_J',100)   TARGET:J;
    6               ASRT: S1.TOPZ = S.TOPZ + 1   TARGET:S1.TOPZ
    6                              SOURCE: S.TOPZ;
    7               ASRT: IF J=S.TOPZ THEN S1.Z(J) = X
    7                  ELSE S1.Z(J) = S.Z(J)   TARGET:S1.Z(J)
    7                              SOURCE: S.TOPZ,S.Z(J);
    8               ASRT: IF J=S.TOPZ THEN END_J(J) = TRUE
    8                                 ELSE END_J(J) = FALSE
    8                       TARGET:END_J(J)
    8                       SOURCE:S.TOPZ;
    9         MODFUN POP(S:S STACK) RETURNS(S1: STACK);
   10            STIM;
   11               ASRT: J = SUBS('S.Z, S1.Z,END_J',100)   TARGET:J;
   12               ASRT: S1.TOPZ = S.TOPZ - 1   TARGET:S1.TOPZ
   12                              SOURCE: S.TOPZ;
   13               ASRT: S1.Z(J) = S.Z(J)  TARGET:S1.Z(J)
   13                     SOURCE: S.Z(J);
   14               ASRT: IF J=S.TOPZ THEN END_J(J) = TRUE
   14                                 ELSE END_J(J) = FALSE
   14                       TARGET:END_J(J)
   14                       SOURCE:S.TOPZ;
   15            LOGIC: |DUM;

   16         MODFUN TOP(S:S STACK) RETURNS(Y: INTEGER);
   17            STIM;
   18               ASRT: J = SUBS('S.Z',100)   TARGET: J;
   19               ASRT: Y = S.Z (S.TOPZ)    TARGET: Y
   19                  SOURCE: S.TOPZ,S.Z(S.TOPZ);
   20            LOGIC: |DUM;

   21         MODFUN EMPTYSTACK RETURNS(S1: STACK);
   22            STIM;
   23               ASRT: S1.TOPZ = 0    TARGET:S1.TOPZ;
   24            LOGIC: |DUM;


   25         DIAG DUM: PRINT = MSG;
   26         MESSAGE MSG: ' NOTHING ';
   27         FUNCTION SUBS, TYPE=E;
   28         FUNCTION TRUE, TYPE=E, VALUE=BIT(1);
   29         FUNCTION FALSE, TYPE=E, VALUE=BIT(1);
   30      END STACK;
```

```
                              INTRA TEST SEQUENCING PUSH
                              ANALYSIS OF THE ADJACENCY MATRIX
                                                        1 1 1 1 1 1 1
                                      1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
                                      - - - - - - - - - - - - - - - -
  1  $S_#0001      ASSERTION          0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
  2  $S_#0002      ASSERTION          0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0
  3  $S_#0003      ASSERTION          0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
  4  $S_#0004      ASSERTION          0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
  5  SUBS          VARIABLE           1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  6  J             VARIABLE           0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
  7  S.TOPZ        VARIABLE           0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
  8  S1.TOPZ       VARIABLE           0 0 0 0 0 0 0 0 0 0 0 0 0 019
  9  X             VARIABLE           0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 10  S.Z           VARIABLE           0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 11  S1.Z          VARIABLE           0 0 0 0 0 0 0 0 0 0 0 0 0 019
 12  TRUE          VARIABLE           0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 13  FALSE         VARIABLE           0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 14  END_J         VARIABLE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 15  S             VARIABLE           0 0 0 0 0 019 0 019 0 0 0 0 0 0
 16  S1            VARIABLE           0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                              SEQUENCE OF PROCESSING FOR TEST PUSH
INDER
VECT  ORDER   RANK         NAME            TYPE              TEXT
INDEX VECTOR
  1     5       0          SUBS            VARIABLE          GLOBAL / SOURCE /

  2     9       0          X               VARIABLE          LOCAL

  3    12       0          TRUE            VARIABLE          GLOBAL / SOURCE /

  4    13       0          FALSE           VARIABLE          GLOBAL / SOURCE /

  5    15       0          S               VARIABLE          LOCAL

  6     1       1          $S_#0001        ASSERTION         J =  SUBS('S.Z, S1.Z,END_J',100)
                                                             TARGET:  J
                                                             SOURCE:  SUBS;
 7     7     1       S.TOPZ        VARIABLE         GLOBAL / SOURCE /

 8     2     2       $S_W00C2      ASSERTION        S1.TOPZ =  S.TOPZ+1
                                                    TARGET:  S1.TOPZ
                                                    SOURCE:  S.TOPZ;

 9     6     2       J             VARIABLE         GLOBAL / TARGET /

10     8     3       S1.TOPZ       VARIABLE         GLOBAL / TARGET /

 LOOP-1 STARTS; SUBSCRIPT J ITERATES FROM 1 TO   100
11     4     4       $S_W00C4      ASSERTION        IF J=S.TOPZ THEN END_J(J) =  TRUE
                                                    ELSE END_J(J) =  FALSE
                                                    TARGET:  END_J
                                                    SOURCE:  J,  S.TOPZ,  TRUE,  FALSE;

12    10     4       S.Z           SUBSCRIPT_VA     GLOBAL / SOURCE /

13     3     5       $S_W00C3      ASSERTION        IF J=S.TOPZ THEN S1.Z(J) =  X
                                                    ELSE S1.Z(J) =  S.Z(J)
                                                    TARGET:  S1.Z
                                                    SOURCE:  J,  S.TOPZ,  X,  S.Z;

14    14     5       END_J         SUBSCRIPT_VA     GLOBAL / TARGET /

15    11     6       S1.Z          SUBSCRIPT_VA     GLOBAL / TARGET /

16    16     7       S1            VARIABLE         LOCAL
 LOOP-1 ENDS;


LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS   11      LAST NODE IS   16      SUBSCRIPT IS J


                          INTRA TEST SEQUENCING POP
                          ANALYSIS OF THE ADJACENCY MATRIX
                                                 1 1 1 1 1 1 1
                               1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6
                               - - - - - - - - - - - - - - - -
   1  SS_W0001     ASSERTION     0 0 0 017 0 1 0 0 0 0 0 0 0 0 0
   2  SS_W0002     ASSERTION     0 0 0 017 0 0 0 1 0 0 0 0 0 0 0
   3  SS_W0003     ASSERTION     0 0 0 017 0 0 0 0 0 1 0 0 0 0 0
   4  SS_W0004     ASSERTION     0 0 0 017 0 0 0 0 0 0 0 0 1 0 0
   5  DUM          DIAGNOSES     0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   6  SUBS         VARIABLE      1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
   7  J            VARIABLE      0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
   8  S.TOPZ       VARIABLE      0 1 0 1 0 0 0 0 0 0 0 0 0 0 0
   9  S1.TOPZ      VARIABLE      0 0 0 0 0 0 0 0 0 0 0 0 0 019
  10  S.Z          VARIABLE      0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
  11  S1.Z         VARIABLE      0 0 0 0 0 0 0 0 0 0 0 0 0 019
  12  TRUE         VARIABLE      0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
  13  FALSE        VARIABLE      0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
  14  END_J        VARIABLE      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  15  S            VARIABLE      0 0 0 0 0 0 019 019 0 0 0 0 0 0
  16  S1           VARIABLE      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                            SEQUENCE OF PROCESSING FOR TEST POP
ORDER
VECT  ORDER    RANK        NAME           TYPE              TEXT
INDEX VECTOR
  1      6       0         SUBS           VARIABLE          GLOBAL / SOURCE /


  2     12       0         TRUE           VARIABLE          GLOBAL / SOURCE /


  3     13       0         FALSE          VARIABLE          GLOBAL / SOURCE /


  4     15       0         S              VARIABLE          LOCAL


  5      1       1         SS_W0001       ASSERTION         J =  SUBS('S.Z, S1.Z,END_J',100)
                                                            TARGET:  J
                                                            SOURCE:  SUBS;

  6      8       1         S.TOPZ         VARIABLE          GLOBAL / SOURCE /


  7      2       2         SS_W0002       ASSERTION         S1.TOPZ =  S.TOPZ-1
                                                            TARGET:  S1.TOPZ
                                                            SOURCE:  S.TOPZ;

  8      7       2         J              VARIABLE          GLOBAL / TARGET /


  9      9       3         S1.TOPZ        VARIABLE          GLOBAL / TARGET /


 LOOP-1 STARTS: SUBSCRIPT J ITERATES FROM 1 TO   100
 10      4       4         SS_W0004       ASSERTION         IF J=S.TOPZ THEN END_J(J) =  TRUE
```

```
                                                         ELSE END_J(J) =  FALSE
                                                         TARGET: END_J
                                                         SOURCE: J,  S.TOPZ,  TRUE,  FALSE;

  11    10     4        S.Z            SUBSCRIPT_VA     GLOBAL / SOURCE /

  12     3     5        SS_W0003       ASSERTION        S1.Z(J) =  S.Z(J)
                                                         TARGET:  S1.Z
                                                         SOURCE:  J,  S.Z;

  13    14     5        END_J          SUBSCRIPT_VA     GLOBAL / TARGET /

  14     5     6        DUM            DIAGNOSES        PRINT = MSG;

  15    11     6        S1.Z           SUBSCRIPT_VA     GLOBAL / TARGET /

  16    16     7        S1             VARIABLE         LOCAL
   LOOP-1 ENDS;


 LOOP SUMMARY TABLE :
    LOOP-1     FIRST NODE IS    10      LAST NODE IS   16      SUBSCRIPT IS J


                               INTRA TEST SEQUENCING TOP
                               ANALYSIS OF THE ADJACENCY MATRIX
                                          1 2 3 4 5 6 7 8 9
                                          - - - - - - - - -
    1  SS_W0001      ASSERTION       C 017 0 1 0 0 0 0
    2  SS_W0002      ASSERTION       0 017 0 0 0 0 1 0
    3  DUM           DIAGNOSES       0 0 0 0 0 0 0 0 0
    4  SUBS          VARIABLE        1 0 0 0 0 0 0 0 0
    5  J             VARIABLE        0 C C 0 C 0 0 0 C
    6  S.Z           VARIABLE        0 1 0 0 0 0 0 0 0
    7  S.TOPZ        VARIABLE        0 1 0 0 0 0 0 0 0
    8  V             VARIABLE        0 0 0 0 0 0 0 0 0
    9  S             VARIABLE        0 0 0 0 01919 0 C
                               SEQUENCE OF PROCESSING FOR TEST TOP
ORDER
VECT  ORDER   RANK       NAME          TYPE                   TEXT
INDEX VECTOR
   1     4      0        SUBS          VARIABLE         GLOBAL / SOURCE /
  2     9     0       S               VARIABLE        LOCAL
  3     1     1       $S_W0001        ASSERTION       J =  SUBS('S.Z',100)
                                                      TARGET:  J
                                                      SOURCE:  SUBS;
  4     7     1       S.TOPZ          VARIABLE        GLOBAL / SOURCE /
  5     5     2       J               VARIABLE        GLOBAL / TARGET /
  6     6     2       S.Z             SUBSCRIPT_VA    GLOBAL / SOURCE /
  7     2     3       $S_W0002        ASSERTION       Y =  S.Z(S.TOPZ)
                                                      TARGET:  Y
                                                      SOURCE:  S.Z,  S.TOPZ;
  8     3     4       DUM             DIAGNOSES       PRINT = MSG;
  9     8     4       Y               VARIABLE        LOCAL


                         INTRA TEST SEQUENCING EMPTYSTACK
                         ANALYSIS OF THE ADJACENCY MATRIX
                                        1 2 3 4
                                        - - - -
    1  $S_W0001      ASSERTION          017 1 0
    2  DUM           DIAGNOSES          0 0 0 0
    3  S1.TOPZ       VARIABLE           0 0 019
    4  S1            VARIABLE           0 0 0 0
                         SEQUENCE OF PROCESSING FOR TEST EMPTYSTACK
ORDER
VECT  ORDER    RANK       NAME            TYPE                  TEXT
INDEX VECTOR
  1     1       0       $S_W0001        ASSERTION        S1.TOPZ =  0
                                                         TARGET:  S1.TOPZ;

  2     2       1       DUM             DIAGNOSES        PRINT = MSG;

  3     3       1       S1.TOPZ         VARIABLE         GLOBAL / TARGET /

  4     4       2       S1              VARIABLE         LOCAL
```

```
C**************************************************************************$
C**************************************************************************$
C    BEGIN EQUATE PROGRAM 'STACK' $
C**************************************************************************$
C**************************************************************************$
C             NAMES                     INDEX
C             *** TEST MODULES ***
C             'PUSH'              --  1
C             'POP'               --  2
C             'TOP'               --  3
C             'EMPTYSTACK'        --  4
C
C             *** DIAGNOSES ***
C             'DUM'               --  1
        DECLARE DIGITAL, 'SYS.VIRTITER.STA'  $
C    UUT POINT DEFINITIONS      $
C    DECLARATIONS FOR USER DEFINED GLOBAL VARIABLES   $
        DECLARE DECIMAL,  'J.STA'  $
        DECLARE DECIMAL, LIST, 'TOPZ.STA' (100) $
        DECLARE DECIMAL, LIST, 'TOPZ.STA' (100) $
        DECLARE DECIMAL, LIST, 'Z.STA' (100,100) $
        DECLARE DECIMAL, LIST, 'Z.STA' (100,100) $
        DECLARE DIGITAL, LIST, 'END-J.STA' (100) $
        DECLARE DECIMAL, 'X.1.STA'  $
        DECLARE DECIMAL, 'S.1.STA'  $
        DECLARE DECIMAL, 'S1.1.STA'  $
        DECLARE DIGITAL, 'SYS.DIG.01.STA' $
        DECLARE DIGITAL, 'SYS.DIG.02.STA' $
        DECLARE DECIMAL, 'S.2.STA'  $
        DECLARE DECIMAL, 'S1.2.STA'  $
        DECLARE DECIMAL, 'Y.3.STA'  $
        DECLARE DECIMAL, 'S.3.STA'  $
        DECLARE DECIMAL, 'S1.4.STA'  $
C $
C    USER DEFINED ATE FUNCTIONS $
C $
        INCLUDE "UPFLIB" $
C**************************************************************************$
C**************************************************************************$
C    DIAGNOSES PRCCS $
C**************************************************************************$
C**************************************************************************$
 6000   DEFINE PROCEDURE, 'DUM.STA' $
        RECORD " NOTHING " $
        END 'DUM.STA' $
C**************************************************************************$
C**************************************************************************$
C    TEST PROCS $
C**************************************************************************$
C**************************************************************************$
 6100    DEFINE PROCEDURE, 'PUSH' $
        'X.1.STA' = 'PUSH.PRM03' $
        'S.1.STA' = 'PUSH.PRMC2' $
        'STACK.GEN' = 'STACK.GEN' + 1  $
        'S1' = 'STACK.GEN' $
        'SYS.FLAG.STA' = 'SYS.TRUE' $
        'SYS.ASRT-FLAG.STA' = 'SYS.TRUE' $
        'TOPZ.STA'('S1.1.STA') = ('TOPZ.STA'('S.1.STA')+ 1) $
        'END-J.STA' = 'SYS.TRUE' $
        FOR 'J.1.STA' = 1 THRU 100 BY 1 THEN $
            'SYS.ASRT-FLAG.STA' = 'SYS.TRUE' $
            COMPARE 'J.STA', EQ 'TOPZ.STA'('S.1.STA') $
```

```
                 GOTO STEP  6105     IF NOGO $
                 'SYS.DIG.01.STA' = 'SYS.TRUE' $
                 GOTO STEP  6110       $
6105             'SYS.DIG.01.STA' = 'SYS.FALSE' $
6110             COMPARE 'SYS.DIG.01.STA', EQ 'SYS.TRUE' $
                 GOTO STEP  6115     IF NOGO $
                 'END-J.STA'('J.STA') = 'SYS.TRUE' $
                 GOTO STEP  6120       $
6115             'END-J.STA'('J.STA') = 'SYS.FALSE' $
6120             COMPARE 'J.STA', EQ 'TOPZ.STA'('S.1.STA') $
                 GOTO STEP  6125     IF NOGO $
                 'SYS.DIG.02.STA' = 'SYS.TRUE' $
                 GOTO STEP  6130       $
6125             'SYS.DIG.02.STA' = 'SYS.FALSE' $
6130             COMPARE 'SYS.DIG.02.STA', EQ 'SYS.TRUE' $
                 GOTO STEP  6135     IF NOGO $
                 'Z.STA'('S1.1.STA','J.STA') = 'X.1.STA' $
                 GOTO STEP  6140       $
6135             'Z.STA'('S1.1.STA','J.STA') = 'Z.STA'('S.1.STA','J.STA') $
6140             COMPARE 'END-J.STA', EQ  'SYS.TRUE' $
                 GOTO STEP  6145     IF NOGO $
             END FOR $
6145         'PUSH.RES' = 'S1' $
             END 'PUSH' $
C************************************************************************$
6200         DEFINE PROCEDURE, 'POP' $
             'S.2.STA' = 'POP.PRM02' $
             'STACK.GEN' = 'STACK.GEN' + 1  $
             'S1' = 'STACK.GEN' $
             'SYS.FLAG.STA' = 'SYS.TRUE' $
             'SYS.ASRT-FLAG.STA' = 'SYS.TRUE' $
             'TOPZ.STA'('S1.2.STA') = ('TOPZ.STA'('S.2.STA')- 1) $
             'END-J.STA' = 'SYS.TRUE' $
             FOR 'J.2.STA' = 1 THRU 100 BY 1 THEN $
                 'SYS.ASRT-FLAG.STA' = 'SYS.TRUE' $
                 COMPARE 'J.STA', EQ 'TOPZ.STA'('S.2.STA') $
                 GOTO STEP  6205     IF NOGO $
                 'SYS.DIG.01.STA' = 'SYS.TRUE' $
                 GOTO STEP  6210       $
6205             'SYS.DIG.01.STA' = 'SYS.FALSE' $
6210             COMPARE 'SYS.DIG.01.STA', EQ 'SYS.TRUE' $
                 GOTO STEP  6215     IF NOGO $
                 'END-J.STA'('J.STA') = 'SYS.TRUE' $
                 GOTO STEP  6220       $
6215             'END-J.STA'('J.STA') = 'SYS.FALSE' $
6220             'Z.STA'('S1.2.STA','J.STA') = 'Z.STA'('S.2.STA','J.STA') $
6225             COMPARE 'SYS.ASRT-FLAG.STA', EQ 'SYS.TRUE' $
                 GOTO STEP  6230     IF NOGO $
                 PERFORM 'DUM.STA'  $
6230             COMPARE 'END-J.STA', EQ  'SYS.TRUE' $
                 GOTO STEP  6235     IF NOGO $
             END FOR $
6235         'POP.RES' = 'S1' $
             END 'POP' $
```

```
C*********************************************************************$
 6300      DEFINE PROCEDURE, 'TOP' $
           'S.3.STA' = 'TOP.PRM02' $
           'SYS.FLAG.STA' = 'SYS.TRUE' $
           'SYS.ASRT-FLAG.STA' = 'SYS.TRUE' $
           'Y.3.STA' = 'Z.STA'('S.3.STA','TOPZ.STA'('S.3.STA')) $
 6305      COMPARE 'SYS.ASRT-FLAG.STA', EQ 'SYS.TRUE' $
           GOTO STEP  6310      IF NOGO $
           PERFORM 'DUM.STA'  $
 6310      'TOP.RES' = 'Y' $
           END 'TOP' $
C*********************************************************************$
 6400      DEFINE PROCEDURE, 'EMPTYSTACK' $
           'STACK.GEN' = 'STACK.GEN' + 1  $
           'S1' = 'STACK.GEN' $
           'SYS.FLAG.STA' = 'SYS.TRUE' $
           'SYS.ASRT-FLAG.STA' = 'SYS.TRUE' $
           'TOPZ.STA'('S1.4.STA') = 0 $
 6405      COMPARE 'SYS.ASRT-FLAG.STA', EQ 'SYS.TRUE' $
           GOTO STEP  6410      IF NOGO $
           PERFORM 'DUM.STA'  $
 6410      'EMPTYSTACK.RES' = 'S1' $
           END 'EMPTYSTACK' $
C*********************************************************************$
C*********************************************************************$
C  TRMNATE EQUATE ATLAS PROGRAM. $
C*********************************************************************$
C*********************************************************************$
```

```
1                                    CROSS REFERENCE AND ATTRIBUTES REPORT

 NAME                DEF NO.          ATTRIBUTES AND REFERENCES
 ----                -------          -------------------------
 BIT                   --             DATA TYPE
                                        26  29
 DUM                   25             DIAGNOSIS LABEL
                                        15  20  24
 EMPTYSTACK            21             TEST LABEL
                                        22  24
 END_J                  8             VARIABLE ID, 1-DIMENSIONAL ARRAY, GLOBAL
 END_J                 14             VARIABLE ID, 1-DIMENSIONAL ARRAY, GLOBAL
 FALSE                 29             ATE-FUNCTION ID, E
                                        14   8
 INTEGER               --             DATA TYPE
                                         2   3  16
 J                      5             VARIABLE ID, GLOBAL
                                         7   8  13  14
 J                     11             VARIABLE ID, GLOBAL
                                         7   8  13  14
 J                     18             VARIABLE ID, GLOBAL
                                         7   8  13  14
 MSG                   26             MESSAGE LABEL
                                         7   8  13  14  25
 POP                    9             TEST LABEL

 PUSH                   3             TEST LABEL
                                         4
 RECORD                --             DATA TYPE
                                         2
 S                      3             VARIABLE ID, GLOBAL
 S                      9             VARIABLE ID, GLOBAL
 S                     16             VARIABLE ID, GLOBAL
 STACK                  1             SPECIFICATION LABEL
                                        19  13   7  30
 STACK                 --             DATA TYPE
                                         3   9  16  21
 STACK                  2             VARIABLE ID, GLOBAL
 SUBS                  27             ATE-FUNCTION ID, E
                                        18  11   5
 S1                     3             VARIABLE ID, GLOBAL
 S1                     9             VARIABLE ID, GLOBAL
 S1                    21             VARIABLE ID, GLOBAL
 S1.TOPZ                6             VARIABLE ID, GLOBAL
 S1.TOPZ               12             VARIABLE ID, GLOBAL
 S1.TOPZ               23             VARIABLE ID, GLOBAL
 S1.Z                   7             VARIABLE ID, 1-DIMENSIONAL ARRAY, GLOBAL
 S1.Z                  13             VARIABLE ID, 1-DIMENSIONAL ARRAY, GLOBAL
 TOP                   16             TEST LABEL
                                        17  20
 TOPZ                   2             VARIABLE ID, GLOBAL
 TRUE                  28             ATE-FUNCTION ID, E
                                        14   8
 X                      3             VARIABLE ID
                                         7
 Y                     16             VARIABLE ID
1                                       19
 Y                     19             VARIABLE ID
                                        16
 Z                      2             VARIABLE ID, 1-DIMENSIONAL ARRAY, GLOBAL
```

```
ERROR/WARNING MESSAGES GENERATED          DURING NOPAL SYNTAX ANALYSIS:
 *STATISTICS* NO. OF SAP ERRORS =    0  ,   NO. OF WARNINGS =    0  , NO. OF STATEMENTS=   30



ERROR/WARNING MESSAGES GENERATED DURING CROSS-REFERENCE:


*STATISTICS*  NO. OF XREFT ERRORS =    0     NO. OF WARNINGS =    0



 ERROR/WARNING MESSAGES GENERATED DURING SEQUENCING AND CODE GENERATION:

** WARNING ** (POSSIBLE INCOMPLETENESS): GLOBAL VAR J DEFINED AS TARGET IN TEST PUSH; BUT NEVER USED.

** WARNING ** (POSSIBLE INCOMPLETENESS): GLOBAL VAR J DEFINED AS TARGET IN TEST POP; BUT NEVER USED.

** WARNING ** (POSSIBLE INCOMPLETENESS): GLOBAL VAR J DEFINED AS TARGET IN TEST TOP; BUT NEVER USED.

 ** WARNING ** (POSSIBLE AMBIGUITY):  GLOBAL VAR J DEFINED AS TARGET MORE THAN ONCE IN: TEST PUSH, TEST POP, TEST TOP TH
EY MUST BE UNDER MUTUALLY EXCLUSIVE CONDITION.

 ** WARNING ** (POSSIBLE AMBIGUITY):  GLOBAL VAR END_J DEFINED AS TARGET MORE THAN ONCE IN: TEST PUSH, TEST POP THEY MUS
T BE UNDER MUTUALLY EXCLUSIVE CONDITION.


WARNING MESSAGES GENERATED DURING CODE GENERATION
*WARNING* UPFLIB DOES NOT EXIST  SHOULD BE INCLUDED AT ATLAS COMPILE TIME.
 *WARNING(POSSIBLE INCONSISTENT): IN STATEMENT NUMBER     19
            1 -TH SUBSCRIPT OF VARIABLE S.Z IS A SUBSCRIPTED VARIABLE OR A NON-FREE SUBSCRIPT; RANGE TEST IS NOT PERFORM
ED.
  *STATISTICS* NO. CODE GENERATION ERRORS =    0, NO. OF WARNINGS=    1

  *STATISTICS* NO. OF SEQUENCING ERRORS=    0, NO. OF WARNINGS=    6
```

A.2 ACKERMANN'S FUNCTION

Nopal specification which specifies Ackermann's function is given in this section. Ackermann's function as expressed by recursive equations is:

$$A(0,n) = n+1 \qquad (A\text{-}1)$$
$$A(m,0) = A(m-1,1) \qquad (A\text{-}2)$$
$$A(m,n) = A(m-1,A(m,n-1)) \qquad (A\text{-}3)$$

Nopal specification is based on the simulation of function calls by means of a stack. It is convenient to imagine that there is an array V of stacks, which is represented in the specification by arrays TOP, LBO, and S. An element V(I) of the array of stacks is represented by LAST(I) which gives the top of the stack V(I), LBO(I) which gives the second element of the stack, and S(I) which gives the rest of the stack.

COMPUTE specifies the computation of the value of Ackermann's function with arguments M, N. To begin, values of M and N are placed on the stack with N being on the top; this is done by defining the value of LAST(1) to be equal to N, the value of LBO(1) equal to M, and the value of S(1) to be a stack with a special symbol -1 signifying the bottom of the stack V(1). Top two elements of the stack V, always contain the arguments for which Ackermann's function needs to be computed at any point in execution. Finally, a single value is left on the stack, and that gives the value of Ackermann's function for arguments M, N. The assertions given by statements 15, 16, and 17 can now be explained as follows:

If the top element of stack V(I-1), i.e. LAST(I-1), is equal zero then it corresponds to Equation A-2. The top two elements (p,0) of stack V(I-1) should be replaced by (p-1,1). This is accomplished in the specification by defining LAST(I) to be 1, LBO(I) to be (LBO(I-1) - 1), and S(I) to be S(I-1).

Similarly, if the second element (from the top) of the stack V(I-1) is zero, it corresponds to

Equation A-1, and the Ackermann's function for the top two elements of the stack evaluates to one plus the top element of the stack. This means popping the stack twice, and pushing the new value on the stack. Thus in the specification, if LBO(I-1) is zero then LAST(I) is equal to (LAST(I) + 1), LBO(I) is equal to TOP(S(I-1)), and S(I) is equal to POP(S(I-1)).

Finally, if it is none of the above two cases, action corresponding to the RHS of Equation A-3 is carried out.

After the specification, sequencing reports are included; they are: inter-test and intra-test sequencing reports. Inter-test sequencing report shows that the test INIT should be performed before the test COMPUTE. Intra-test reports contain sequencing of tests INIT and COMPUTE.

```
        /* NOPAL TEST SPECIFICATION SOURCE INPUT, FILE: SAPLIST */

    NOPAL PROCESSOR OPTIONS SPECIFIED: SAPLIST,XREF1,CODE,SEQ,NOXREF2,SOURCE2,TRACE=3,DEBUG=SEQ.CODE


    STMT NO.
    --------

       1         NOPAL MAIN ACKERMANN;
       2            DCL S:STACK ARRAY(*);
       3            DCL NEW: STACK;
                     /* TWO TOP ELEMENTS: LAST AND LBO (I.E. LAST BUT ONE) */
       4            DCL LBO: INTEGER ARRAY(*);
       5            DCL LAST: INTEGER ARRAY(*);

       6            TEST INIT;
       7               STIM;
       8                  ASRT: N=3     TARGET: N;
       9                  ASRT: M=2     TARGET: M;
      10                  ASRT: NEW = NEWSTACK     TARGET: NEW;
      11               LOGIC: *DUM;



      12            TEST COMPUTE;
      13               STIM;

      14                  ASRT: I = SUBS('LAST,LBO,END_I,S',*)    TARGET: I;


      15                  ASRT: IF I=1 THEN LAST(I) = N
      15                     ELSE IF LBO(I-1) = 0 THEN LAST(I) = LAST(I-1) + 1
      15                       ELSE IF LAST(I-1) = 0 THEN LAST(I) = 1
      15                          ELSE LAST(I) = LAST(I-1) - 1
      15                  TARGET: LAST(I)
      15                  SOURCE: LAST(I-1),LBO(I-1);


      16                  ASRT: IF I=1 THEN LBO(I) = M
      16                     ELSE IF LBO(I-1) = 0 THEN LBO(I) = TOP(S(I-1))
      16                       ELSE IF LAST(I-1) = 0 THEN LBO(I) = LBO(I-1) - 1
      16                          ELSE LBO(I) = LBO(I-1)
      16                  TARGET: LBO(I)
      16                  SOURCE: S(I-1),LBO(I-1),LAST(I-1);


      17                  ASRT: IF I=1 THEN S(I) = PUSH(NEW, -1)
      17                     ELSE IF LBO(I-1) = 0 THEN S(I) = POP(S(I-1))
      17                       ELSE IF LAST(I-1) = 0 THEN S(I) = S(I-1)
      17                          ELSE S(I) = PUSH(S(I-1),LBO(I-1)-1)
      17                  TARGET: S(I)
```

```
 17              SOURCE: S(I-1),LBO(I-1),LAST(I-1);

 18              ASRT: IF LBO(I) = -1 THEN END_I(I) = TRUE
 18                                   ELSE END_I(I) = FALSE
 18                 TARGET: END_I(I)
 18                 SOURCE: LBO(I);

 19           LOGIC: *DUM;


 20        DIAG DUM: PRINT = MSG;
 21        MESSAGE MSG: " NOTHING ";

 22        FUNCTION SUBS, TYPE=E;
 23        FUNCTION TRUE, TYPE=E, VALUE=BIT;
 24        FUNCTION FALSE, TYPE=E, VALUE=BIT;
 25        FUNCTION PUSH, TYPE=E, PARAM=(ST,S STACK), PARAM=(X,S INTEGER),
 25              VALUE=STACK;
 26        FUNCTION POP,  TYPE=E, PARAM=(ST,S STACK), VALUE=STACK;
 27        FUNCTION TOP, TYPE=E, PARAM=(ST,S STACK), VALUE =INTEGER;
 28        FUNCTION NEWSTACK, TYPE=E, VALUE=STACK;
 29     END ACKERMANN;
```

```
                            WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION ACKERMANN                              PAGE   1

                                                                           1   1   1   1   1   1   1   1
                                          1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   6   7
                                        ------------------------------------------------------------------
  1   INIT              TEST        |   0   0  11   1   1   0   1   0   0   0   0   0   0   0   0   0   0
  2   COMPUTE           TEST        |   0   0  11   0   0   0   0   0   1   0   1   0   0   1   0   0   1
  3   DUM               DIAGNOSIS   |   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  4   N                 VARIABLE    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  5   M                 VARIABLE    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  6   NEWSTACK          FUNCTION    |   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  7   NEW               VARIABLE    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  8   SUBS              VARIABLE    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  9   LAST              VARIABLE    |   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 10   TOP               FUNCTION    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
                            WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION ACKERMANN                              PAGE   1

                                                                           1   1   1   1   1   1   1   1
                                          1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   6   7
                                        ------------------------------------------------------------------
 11   LBO               VARIABLE    |   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 12   PUSH              FUNCTION    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 13   POP               FUNCTION    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 14   S                 VARIABLE    |   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 15   TRUE              FUNCTION    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
 16   FALSE             FUNCTION    |   0   1   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

```
   17   END_1               VARIABLE 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                         SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION ACKERMANN                    PAGE    1

ORDER
VECTOR  ORDER    RANK         TYPE                        NAME
INDEX   VECTOR
   1       6       0    EVAL OR CONTROL FUNCTION     NEWSTACK
   2       8       0    ITERATION VARIABLE           SUBS
   3      10       0    EVAL OR CONTROL FUNCTION     TOP
   4      12       0    EVAL OR CONTROL FUNCTION     PUSH
   5      13       0    EVAL OR CONTROL FUNCTION     POP
   6      15       0    EVAL OR CONTROL FUNCTION     TRUE
   7      16       0    EVAL OR CONTROL FUNCTION     FALSE
   8       1       1    TEST                         INIT
   9       4       2    GLOBAL VARIABLE              N
  10       5       2    GLOBAL VARIABLE              M
                         SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION ACKERMANN                    PAGE    1

ORDER
VECTOR  ORDER    RANK         TYPE                        NAME
INDEX   VECTOR
  11       7       2    GLOBAL VARIABLE              NEW
  12       2       3    TEST                         COMPUTE
  13       3       4    DIAGNOSIS                    DUM
  14       9       4    GLOBAL VARIABLE              LAST
  15      11       4    GLOBAL VARIABLE              LBO
  16      14       4    GLOBAL VARIABLE              S
  17      17       4    GLOBAL VARIABLE              END_I


                         INTRA TEST SEQUENCING INIT
                         ANALYSIS OF THE ADJACENCY MATRIX
                                 1 2 3 4 5 6 7 8
                                 - - - - - - - -
   1  $S_W0001      ASSERTION     0 0 0 0 1 0 0 0
   2  $S_W0002      ASSERTION     0 0 0 0 0 1 0 0
   3  $S_W0003      ASSERTION     0 0 0 0 0 0 0 1
   4  DUM           DIAGNOSES    161616 0 0 0 0 0
   5  N             VARIABLE      0 0 0 0 0 0 0 0
   6  M             VARIABLE      0 0 0 0 0 0 0 0
   7  NEWSTACK      VARIABLE      0 0 1 0 0 0 0 0
   8  NEW           VARIABLE      0 0 0 0 0 0 0 0
                         SEQUENCE OF PROCESSING FOR TEST INIT
ORDER
VECT  ORDER   RANK      NAME            TYPE                 TEXT
INDEX VECTOR
  1     4      0       DUM            DIAGNOSES        PRINT = MSG;

  2     7      0       NEWSTACK       VARIABLE         GLOBAL / SOURCE /

  3     1      1       $S_W0001       ASSERTION        N =  3
```

```
                                                               TARGET:  N;
    4      2       1        $S_W0002        ASSERTION          M =  2
                                                               TARGET:  M;

    5      3       1        $S_W00C3        ASSERTION          NEW =  NEWSTACK
                                                               TARGET:  NEW
                                                               SOURCE:  NEWSTACK;

    6      5       2        N               VARIABLE           GLOBAL / TARGET /

    7      6       2        M               VARIABLE           GLOBAL / TARGET /

    8      8       2        NEW             VARIABLE           GLOBAL / TARGET /




                            INTRA TEST SEQUENCING COMPUTE
                            ANALYSIS OF THE ADJACENCY MATRIX
                                                            1 1 1 1 1 1 1 1 1 1 2
                                              1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0
                                              - - - - - - - - - - - - - - - - - - - -
   1  $S_W0001        ASSERTION               0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
   2  $S_W0002        ASSERTION               0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
   3  $S_W0C03        ASSERTION               0 0 0 0 0 0 0 0 0 1 0 C 0 0 0 0 0 0 0 0
   4  $S_W0C04        ASSERTION               0 C C 0 0 0 0 0 0 0 0 C 0 1 0 0 0 0 0 0
   5  $S_W0C05        ASSERTION               0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
   6  DUM             DIAGNOSES               1616161616 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
   7  SUBS            VARIABLE                1 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
   8  I               VARIABLE                0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 C 0 0 0
   9  N               VARIABLE                0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  10  LBO             VARIABLE                021 0 0 1 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  11  LAST            VARIABLE                0 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  12  TOP             VARIABLE                0 0 1 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  13  M               VARIABLE                0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  14  S               VARIABLE                0 021 0 C 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  15  PUSH            VARIABLE                0 C C 1 0 0 0 0 0 0 0 C 0 0 0 0 C 0 0 0
  16  POP             VARIABLE                0 0 C 1 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  17  NEW             VARIABLE                0 0 0 1 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  18  TRUE            VARIABLE                0 0 0 0 1 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  19  FALSE           VARIABLE                0 0 0 0 1 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0
  20  END_I           VARIABLE                0 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0

                            SEQUENCE OF PROCESSING FOR TEST COMPUTE
ORDER
VECT   ORDER    RANK        NAME            TYPE                         TEXT
INDEX  VECTOR
   1     6       C          DUM             DIAGNOSES          PRINT = MSG;


   2     7       0          SUBS            VARIABLE           GLOBAL / SOURCE /
```

```
 3     9     0      N              VARIABLE         GLOBAL / SOURCE /

 4    12     0      TOP            VARIABLE         GLOBAL / SOURCE /

 5    13     0      M              VARIABLE         GLOBAL / SOURCE /

 6    15     0      PUSH           VARIABLE         GLOBAL / SOURCE /

 7    16     0      POP            VARIABLE         GLOBAL / SOURCE /

 8    17     0      NEW            VARIABLE         GLOBAL / SOURCE /

 9    18     0      TRUE           VARIABLE         GLOBAL / SOURCE /

10    19     0      FALSE          VARIABLE         GLOBAL / SOURCE /

11     1     1      $S_W0001       ASSERTION        I =  SUBS('LAST,LBO,END_I,S',*)
                                                    TARGET:  I
                                                    SOURCE:  SUBS;

12     8     2      I              VARIABLE         LOCAL

 LOOP-1 STARTS: SUBSCRIPT I ITERATES FROM 1 TO *
13     4     3      $S_W0004       ASSERTION        IF I=1 THEN S(I) =  PUSH(NEW,-1)
                                                    ELSE      IF LBO(I-1)=0 THEN S(I) =  POP(S(I-1))
                                                    ELSE  IF LAST(I-1)=0 THEN S(I) =  S(I-1)
                                                    ELSE S(I) =  PUSH(S(I-1),LBO(I-1)-1)
                                                    TARGET:  S
                                                    SOURCE:  I,  PUSH,  POP,  NEW;

14    14     4      S              SUBSCRIPT_VA     LOCAL

15     3     5      $S_W0003       ASSERTION        IF I=1 THEN LBO(I) =  M
                                                    ELSE      IF LBO(I-1)=0 THEN LBO(I) =  TOP(S(I-1))
                                                    ELSE  IF LAST(I-1)=0 THEN LBO(I) =  LBO(I-1)-1
                                                    ELSE LBO(I) =  LBO(I-1)
                                                    TARGET:  LBO
                                                    SOURCE:  I,  TOP,  M,  S;

16    10     6      LBO            SUBSCRIPT_VA     LOCAL
```

```
 17     2      7        $S_W0002      ASSERTION       IF I=1 THEN LAST(I) =  N
                                                      ELSE      IF LBO(I-1)=0 THEN LAST(I) =  LAST(I-1)+1
                                                      ELSE  IF LAST(I-1)=0 THEN LAST(I) =  1
                                                      ELSE LAST(I) =  LAST(I-1)-1
                                                      TARGET:  LAST
                                                      SOURCE:  I,  N,  LBO;

 18     5      7        $S_W00C5      ASSERTION       IF LBO(I)=-1 THEN END_I(I) =  TRUE
                                                      ELSE END_I(I) =  FALSE
                                                      TARGET:  END_I
                                                      SOURCE:  I,  LBO,  TRUE,  FALSE;

 19    11      8        LAST          SUBSCRIPT_VA    LOCAL

 20    20      8        END_I         SUBSCRIPT_VA    GLOBAL / TARGET /
  LOOP-1 ENDS;


LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS   13     LAST NODE IS   20     SUBSCRIPT IS I
```

A.3 BAND-WIDTH METER

The main module BWM specifies the application of a voltage source with frequencies in a range with a given step size to a UUT. For each application of a frequency, the gain of voltages (ratio of output voltage to input voltage) across the UUT is measured. A table corresponding to the applied frequencies and the measured gains is printed out. The devices are represented by means of abstract structures: gain measurement device is represented by the structure GD, and frequency source by the structure FS. A call is issued to ACCESS_GAIN_DEVICE whenever the value of GAIN, a field in the structure GD, is accessed (before statement 24); and similarly, a call is issued to SAVE_FREQ_SOURCE in the case of structure FS (after statements 16, 17, and 18). The assertions in the specification are self explanatory.

```
      /* NOPAL TEST SPECIFICATION SOURCE INPUT, FILE: SAPLIST */

NOPAL PROCESSOR OPTIONS SPECIFIED: SAPLIST,XREF1,CODE,SEQ,NOXREF2,SOURCE2,TRACE=3,DEBUG=SEQ


STMT NO.
--------

    1        NOPAL MAIN BWM;

                 /* ABSTRACT STRUCTURES AS DEVICES */
    2           DCL 1 GD: GAIN_DEVICE ARRAY(*),
    2              2 GAIN: REAL;

    3           DCL 1 FS: FREQ_SOURCE ARRAY(*),
    3                2 FREQ: REAL,
    3                2 IN1: INTEGER,
    3                2 IN2: INTEGER;

    4           TEST INIT;
    5              STIM;
    6                 ASRT: FMIN = 100     TARGET: FMIN;
    7                 ASRT: FMAX = 10000     TARGET: FMAX;
    8                 ASRT: I1 = 1       TARGET: I1;
    9                 ASRT: I2 = 2       TARGET: I2;
   10                 ASRT: I3 = 3       TARGET: I3;
   11                 ASRT: I4 = 4       TARGET: I4;
   12              LOGIC:  |DUM;

   13           TEST CHARTFREQ;
   14              STIM;
   15             ASRT: I = SUBS('FREQ,IN1,IN2,PTR1_GD,PTR2_GD,PTR3_GD,PTR4_GD,X,Y,GAIN,ERR'
   15                      ,30)   TARGET: I;
   16                 ASRT: FREQ(I) = FMIN + (FMAX - FMIN)*I/10    TARGET: FREQ(I);
   17                 ASRT: IN1(I) = I1   TARGET: IN1(I);
   18                 ASRT: IN2(I) = I2   TARGET: IN2(I);

                         /* PARAMETERS FOR GAIN DEVICE */
   19                 ASRT: PTR1_GD(I) = I1      TARGET: PTR1_GD(I);
   20                 ASRT: PTR2_GD(I) = I2      TARGET: PTR2_GD(I);
   21                 ASRT: PTR3_GD(I) = I3      TARGET: PTR3_GD(I);
   22                 ASRT: PTR4_GD(I) = I4      TARGET: PTR4_GD(I);

                         /* CAUSES FREQUENCY FREQ(I) TO BE APPLIED. */
   23                 ASRT: X(I) = FREQ(I)     TARGET: X(I)
   23                              SOURCE: FREQ(I);

                         /* CAUSES GAIN MEASUREMENTS TO BE PERFORMED. */
   24                 ASRT: Y(I) = GAIN(I)        TARGET: Y(I)
   24                              SOURCE: GAIN(I);

                         /* CHART: FUNCTION WHICH PRINTS OUT A TABLE. */
   25                 ASRT: ERR(I) = CHART(X(I),Y(I))    TARGET: ERR(I)
   25                              SOURCE: X(I),Y(I);
   26              LOGIC: |DUM;
   27          DIAG DUM: PRINT = MSG;
   28          MESSAGE MSG: ' NOTHING ';
   29          FUNCTION SUBS, TYPE=E;
   30          FUNCTION TRUE, TYPE=E;
   31          FUNCTION CHART, TYPE=E;
   32        END BWM;
```

```
                              WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION BWM                              PAGE  1

                                                                 1  1  1  1  1  1  1  1  1  1  2
                                        1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0
                                      ------------------------------------------------------------
  1   INIT                 TEST      |  0  0 12  1  1  1  1  1  1  0  0  0  0  0  0  0  0  0  0  0
  2   CHARTFREQ            TEST      |  0  0 12  0  0  0  0  0  0  0  1  1  1  1  1  1  1  0  0  1
  3   DUM                  DIAGNOSIS |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  4   FMIN                 VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  5   FMAX                 VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  6   I1                   VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  7   I2                   VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  8   I3                   VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  9   I4                   VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 10   SUBS                 VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                              WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION BWM                              PAGE  1

                                                                 1  1  1  1  1  1  1  1  1  1  2
                                        1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0
                                      ------------------------------------------------------------
 11   FREQ                 VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 12   IN1                  VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 13   IN2                  VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 14   PTR1_GD              VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 15   PTR2_GD              VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 16   PTR3_GD              VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 17   PTR4_GD              VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 18   GAIN                 VARIABLE  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 19   CHART                FUNCTION  |  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 20   ERR                  VARIABLE  |  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                              SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION BWM                                 PAGE  1

ORDER
VECTOR  ORDER    RANK        TYPE                        NAME
INDEX   VECTOR
  1        1      0      TEST                            INIT
  2       10      0      ITERATION VARIABLE              SUBS
  3       18      0      GLOBAL VARIABLE                 GAIN
  4       19      0      EVAL OR CONTROL FUNCTION        CHART
  5        4      1      GLOBAL VARIABLE                 FMIN
  6        5      1      GLOBAL VARIABLE                 FMAX
  7        6      1      GLOBAL VARIABLE                 I1
  8        7      1      GLOBAL VARIABLE                 I2
  9        8      1      GLOBAL VARIABLE                 I3
  10         9         1    GLOBAL VARIABLE                I4
                            SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION BUM                        PAGE   1

 ORDER
 VECTOR   ORDER    RANK          TYPE                      NAME
 INDEX    VECTOR
   11        2       2      TEST                           CHARTFREQ
   12        3       3      DIAGNOSIS                      BUM
   13       11       3      GLOBAL VARIABLE                FREQ
   14       12       3      GLOBAL VARIABLE                IN1
   15       13       3      GLOBAL VARIABLE                IN2
   16       14       3      GLOBAL VARIABLE                PTR1_GD
   17       15       3      GLOBAL VARIABLE                PTR2_GD
   18       16       3      GLOBAL VARIABLE                PTR3_GD
   19       17       3      GLOBAL VARIABLE                PTR4_GD
   20       20       3      GLOBAL VARIABLE                ERR


                            INTRA TEST SEQUENCING INIT
                            ANALYSIS OF THE ADJACENCY MATRIX
                                                     1 1 1 1
                                   1 2 3 4 5 6 7 8 9 0 1 2 3
                                   - - - - - - - - - - - - -
   1  $S_W0001      ASSERTION       0 0 0 0 0 017 1 0 0 0 0 0
   2  $S_W0002      ASSERTION       0 0 0 0 0 017 0 1 0 0 0 0
   3  $S_W0003      ASSERTION       0 0 0 0 0 017 0 0 1 0 0 0
   4  $S_W0004      ASSERTION       0 0 0 0 0 017 0 0 0 1 0 0
   5  $S_W0005      ASSERTION       0 0 0 0 0 017 0 0 0 0 1 0
   6  $S_W0006      ASSERTION       0 0 0 0 0 017 0 0 0 0 0 1
   7  BUM           DIAGNOSES       0 0 0 0 0 0 0 0 0 0 0 0 0
   8  FMIN          VARIABLE        0 0 0 0 0 0 0 0 0 0 0 0 0
   9  FMAX          VARIABLE        0 0 0 0 0 0 0 0 0 0 0 0 0
  10  I1            VARIABLE        0 0 0 0 0 0 0 0 0 0 0 0 0
  11  I2            VARIABLE        0 0 0 0 0 0 0 0 0 0 0 0 0
  12  I3            VARIABLE        0 0 0 0 0 0 0 0 0 0 0 0 0
  13  I4            VARIABLE        0 0 0 0 0 0 0 0 0 0 0 0 0
                            SEQUENCE OF PROCESSING FOR TEST INIT
ORDER
VECT  ORDER    RANK        NAME             TYPE                    TEXT
INDEX VECTOR
  1     1       0        $S_W0001         ASSERTION         FMIN =  100
                                                            TARGET:  FMIN;

  2     2       0        $S_W0002         ASSERTION         FMAX =  10000
                                                            TARGET:  FMAX;

  3     3       0        $S_W0003         ASSERTION         I1 =  1
                                                            TARGET:  I1;

  4     4       0        $S_W0004         ASSERTION         I2 =  2
                                                            TARGET:  I2;
 5     5     0     $S_W0005     ASSERTION     I3 =  3
                                              TARGET:  I3;

 6     6     0     $S_W0006     ASSERTION     I4 =  4
                                              TARGET:  I4;

 7     7     1     DUM          DIAGNOSES     PRINT = MSG;

 8     8     1     FMIN         VARIABLE      GLOBAL / TARGET /

 9     9     1     FMAX         VARIABLE      GLOBAL / TARGET /

10    10     1     I1           VARIABLE      GLOBAL / TARGET /

11    11     1     I2           VARIABLE      GLOBAL / TARGET /

12    12     1     I3           VARIABLE      GLOBAL / TARGET /

13    13     1     I4           VARIABLE      GLOBAL / TARGET /
```

```
                              INTRA TEST SEQUENCING CHARTFREQ
                              ANALYSIS OF THE ADJACENCY MATRIX
                                                          1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3
                                        1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
                                        - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 1  $S_W0001      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 2  $S_W0002      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 3  $S_W0003      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 4  $S_W0004      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
 5  $S_W0005      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
 6  $S_W0006      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
 7  $S_W0007      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
 8  $S_W0008      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
 9  $S_W0009      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
10  $S_W0010      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
11  $S_W0011      ASSERTION             0 0 0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
12  DUM           DIAGNOSES             0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13  SUBS          VARIABLE              1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
14  I             VARIABLE              0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15  FMIN          VARIABLE              0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
16  FMAX          VARIABLE              0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
17  FREQ          VARIABLE              0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0
18  I1            VARIABLE              0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
19  IN1           VARIABLE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0
20  I2            VARIABLE              0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
21  IN2           VARIABLE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0
22  PTR1_GO       VARIABLE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 020
23  PTR2_GO       VARIABLE              0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 020
24  I3            VARIABLE              0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
 25  PTR3_GD        VARIABLE      0 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 020
 26  I4             VARIABLE      0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 27  PTR4_GD        VARIABLE      0 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 020
 28  X              VARIABLE      0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 29  GAIN           VARIABLE      0 0 0 0 0 0 0 0 0 1 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 30  Y              VARIABLE      0 0 0 0 0 0 0 0 0 C 1 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 31  CHART          VARIABLE      0 0 0 0 0 0 0 0 0 0 1 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 32  ERR            VARIABLE      C 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 33  FS             VARIABLE      0 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 34  GD             VARIABLE      0 0 0 0 0 0 0 0 0 0 0 C 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0 0 0 0 0

                        SEQUENCE OF PROCESSING FOR TEST CHARTFREQ
ORDER
VECT  ORDER   RANK       NAME          TYPE                 TEXT
INDEX VECTOR
  1    13      0         SUBS          VARIABLE        GLOBAL / SOURCE /

  2    15      0         FMIN          VARIABLE        GLOBAL / SOURCE /

  3    16      0         FMAX          VARIABLE        GLOBAL / SOURCE /

  4    18      0         I1            VARIABLE        GLOBAL / SOURCE /

  5    20      0         I2            VARIABLE        GLOBAL / SOURCE /

  6    24      0         I3            VARIABLE        GLOBAL / SOURCE /

  7    26      0         I4            VARIABLE        GLOBAL / SOURCE /

  8    31      0         CHART         VARIABLE        GLOBAL / SOURCE /

  9     1      1         $S_*0001      ASSERTION       I
                                                        =  SUBS('FREQ,IN1,IN2,PTR1_GD,PTR2_GD,PTR3_GD,PTR4_GD,X,Y,GAIN,ERR',3
                                                       TARGET:  I
                                                       SOURCE:  SUBS;

 10    14      2         I             VARIABLE        LOCAL

 LOOP-1 STARTS: SUBSCRIPT I ITERATES FROM 1 TO    30
 11     2      3         $S_*0002      ASSERTION       FREQ(I) =  FMIN+(FMAX-FMIN)*I/10
                                                       TARGET:  FREQ
                                                       SOURCE:  I,  FMIN,  FMAX;
```

```
12    3    3    $S_W00C3     ASSERTION       IN1(I) =  I1
                                             TARGET:  IN1
                                             SOURCE:  I,  I1;

13    4    3    $S_W0004     ASSERTION       IN2(I) =  I2
                                             TARGET:  IN2
                                             SOURCE:  I,  I2;

14    5    3    $S_W00C5     ASSERTION       PTR1_GD(I) =  I1
                                             TARGET:  PTR1_GD
                                             SOURCE:  I,  I1;

15    6    3    $S_W00C6     ASSERTION       PTR2_GD(I) =  I2
                                             TARGET:  PTR2_GD
                                             SOURCE:  I,  I2;

16    7    3    $S_W00C7     ASSERTION       PTR3_GD(I) =  I3
                                             TARGET:  PTR3_GD
                                             SOURCE:  I,  I3;

17    8    3    $S_W00C8     ASSERTION       PTR4_GD(I) =  I4
                                             TARGET:  PTR4_GD
                                             SOURCE:  I,  I4;

18   17    4    FREQ         SUBSCRIPT_VA    GLOBAL / TARGET /

19   19    4    IN1          SUBSCRIPT_VA    GLOBAL / TARGET /

20   21    4    IN2          SUBSCRIPT_VA    GLOBAL / TARGET /

21   22    4    PTR1_GD      SUBSCRIPT_VA    GLOBAL / TARGET /

22   23    4    PTR2_GD      SUBSCRIPT_VA    GLOBAL / TARGET /

23   25    4    PTR3_GD      SUBSCRIPT_VA    GLOBAL / TARGET /

24   27    4    PTR4_GD      SUBSCRIPT_VA    GLOBAL / TARGET /

25    9    5    $S_W0009     ASSERTION       X(I) =  FREQ(I)
                                             TARGET:  X
                                             SOURCE:  I,  FREQ;
```

```
 26    33     5      FS          SUBSCRIPT_VA    GLOBAL / SOURCE /

 27    34     5      GD          SUBSCRIPT_VA    GLOBAL / SOURCE /

 28    28     6      X           SUBSCRIPT_VA    LOCAL

 29    29     6      GAIN        SUBSCRIPT_VA    GLOBAL / SOURCE /

 30    10     7      $S_W0010    ASSERTION       Y(I) =  GAIN(I)
                                                 TARGET:  Y
                                                 SOURCE:  I,  GAIN;

 31    30     8      Y           SUBSCRIPT_VA    LOCAL

 32    11     9      $S_W0011    ASSERTION       ERR(I) =  CHART(X(I),Y(I))
                                                 TARGET:  ERR
                                                 SOURCE:  I,  X,  Y,  CHART;

 33    12     10     DUM         DIAGNOSES       PRINT = MSG;

 34    32     10     ERR         SUBSCRIPT_VA    GLOBAL / TARGET /
 LOOP-1 ENDS;

LOOP SUMMARY TABLE :
   LOOP-1     FIRST NODE IS   11     LAST NODE IS   34     SUBSCRIPT IS I
```

The specification for GAIN_DEVICE, given below, has only one modfun called ACCESS_GAIN_DEVICE. The modfun takes voltage measurements across pins (I1,I2) and (I3,I4). The ration of the two measurements defines the value of GAIN. It depends on two funtions: VOLTMETER and MICROVOLTMETER to take the actual measurements.

```
     /* NOPAL TEST SPECIFICATION SOURCE INPUT, FILE: SAPLIST */

NOPAL PROCESSOR OPTIONS SPECIFIED: SAPLIST,XREF1,CODE,SEQ,NOXREF2,SOURCE2,TRACE=3,DEBUG=ALL,LINE=20000


STMT NO.
--------

    1       NOPAL MODULE GAIN_DEVICE;


    2        MODFUN ACCESS_GAIN_DEVICE (I1:S CNX,I2:S CNX,I3:S CNX,I4:S CNX,GAIN:T REAL);
    3             MEAS;

    4                CONJ: (<I1,I2> = MICROVOLTMETER(V1) ) &
    4                      (<I3,I4> = VOLTMETER(V2) )
    4                           TARGET: V1,V2;

    5                ASRT: GAIN = V2 / V1    TARGET: GAIN
    5                                        SOURCE: V1,V2;


    6          LOGIC: |NOTHING;
    7          DIAG NOTHING:
    7             PRINT=MSG1;
    8          MESSAGE MSG1:
    8          TEXT = 'NOTHING';
    9          UUT_PT: I1, ALIAS=I1;
   10          UUT_PT: I2, ALIAS=I2;
   11          UUT_PT: I3, ALIAS=I3;
   12          UUT_PT: I4, ALIAS=I4;
   13          FUNCTION SUBS, TYPE=E;
   14          FUNCTION TRUE, TYPE=E;
   15          FUNCTION VOLTMETER, TYPE=M;
   16          FUNCTION MICROVOLTMETER, TYPE=M;
   17          FUNCTION SUM, TYPE=E;
   18       END GAIN_DEVICE;
```

```
                              INTRA TEST SEQUENCING ACCESS_GAIN_
                              ANALYSIS OF THE ADJACENCY MATRIX
                                                          1
                                           1 2 3 4 5 6 7 8 9 0
                                           - - - - - - - - - -
     1  SM_W0001      CONJUNCTION          0 017 1 1 0 0 0 0 0
     2  SM_W0002      ASSERTION            0 017 0 0 1 0 0 0 0
     3  NOTHING       DIAGNOSES            0 0 0 0 0 0 0 0 0 0
     4  V2            VARIABLE             0 1 0 0 0 0 0 0 0 0
     5  V1            VARIABLE             0 1 0 0 0 0 0 0 0 0
     6  GAIN          VARIABLE             0 0 0 0 0 0 0 0 0 0
     7  I4            VARIABLE             0 0 0 0 0 0 0 0 0 0
     8  I3            VARIABLE             0 0 0 0 0 0 0 0 0 0
     9  I2            VARIABLE             0 0 0 0 0 0 0 0 0 0
    10  I1            VARIABLE             0 0 0 0 0 0 0 0 0 0

                              SEQUENCE OF PROCESSING FOR TEST ACCESS_GAIN_
ORDER
VECT  ORDER   RANK      NAME          TYPE                   TEXT
INDEX VECTOR
  1     1      0      SM_W0001      CONJUNCTION     (<I1, I2>  = MICROVOLTMET(V1)) &(<I3, I4>  = VOLTMETER(V2))
                                                    TARGET:  V2,  V1;

  2     7      0      I4            VARIABLE        LOCAL

  3     8      0      I3            VARIABLE        LOCAL

  4     9      0      I2            VARIABLE        LOCAL

  5    10      0      I1            VARIABLE        LOCAL

  6     4      1      V2            VARIABLE        LOCAL

  7     5      1      V1            VARIABLE        LOCAL

  8     2      2      SM_W0002      ASSERTION       GAIN =  V2/V1
                                                    TARGET:  GAIN
                                                    SOURCE:  V2,  V1;

  9     3      3      NOTHING       DIAGNOSES       PRINT = MSG1;

 10     6      3      GAIN          VARIABLE        LOCAL
```

The specification for FREQ_SOURCE defines a function SAVE_FREQ_SOURCE which specifies the application of a frequency source by means of the function FREQ_GEN. The voltage and frequency to be applied are specified by the *input parameters*. An error is issued if the parameters specify a frequency which falls out of range of the instrument.

```
     /* NOPAL TEST SPECIFICATION SOURCE INPUT, FILE: SAPLIST */

NOPAL PROCESSOR OPTIONS SPECIFIED: SAPLIST,XREF1,CODE,SEQ,NOXREF2,SOURCE2,TRACE=3,DEBUG=SEQ.CODE,LINE=15000


STMT NO.
--------

    1       NOPAL MODULE FREQ_SOURCE;

    2          MODFUN   SAVE_FREQ_SOURCE(FREQ:S REAL,V:S REAL,IN1:S CNX,IN2:S CNX);
    3             STIM;
    4                CONJ: IF RANGE>0 THEN <IN1,IN2> = FREQ_GEN(FREQ,V,RANGE);

    5                ASRT: IF FREQ<F1 & FREQ>=F2 THEN RANGE = 1
    5                   ELSE IF FREQ<F2 & FREQ>F3 THEN RANGE = 2
    5                      ELSE RANGE = 0     TARGET: RANGE;

    6                ASRT: RANGE > 0;

    7                ASRT: F1 = 10000     TARGET:F1;

    8                ASRT: F2 = 100       TARGET:F2;

    9                ASRT: F3 = 10        TARGET: F3;


   10             LOGIC: ∫¯ D1;

   11          DIAG D1:
   11             PRINT = MSG,
   11             PARAMETER = FREQ;
   12          MESSAGE MSG:  TEXT = ´FREQUENCY OUT OF RANGE (P)´;
   13          FUNCTION FREQ_GEN, TYPE=S;
   14          UUT_PT: IN1;
   15          UUT_PT: IN2;
   16       END FREQ_SOURCE;
```

```
                               INTRA TEST SEQUENCING  SAVE _FREQ_
                               ANALYSIS OF THE ADJACENCY MATRIX
                                                         1 1 1 1 1 1
                                         1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
                                         - - - - - - - - - - - - - - -
   1  $S_W0001       CONJUNCTION         0 0 0 0 0 017 0 0 0 0 0 0 0 0
   2  $S_W0002       ASSERTION           0 0 0 0 0 017 0 1 0 0 C 0 0 0
   3  $S_W0003       ASSERTION           0 0 0 0 0 017 0 0 0 0 C 0 0 0
   4  $S_W0004       ASSERTION           0 0 0 0 0 017 0 0 0 0 0 1 0 0
   5  $S_W0005       ASSERTION           0 0 0 0 C 017 0 0 0 0 1 0 0 0
   6  $S_W0006       ASSERTION           0 0 0 0 0 017 0 0 0 1 C 0 0 0
   7  D1             DIAGNOSES           0 0 0 0 0 0 0 0 0 0 0 C 0 0 0
   8  V              VARIABLE            1 0 0 0 0 0 0 0 0 0 0 C 0 0 0
   9  RANGE          VARIABLE            1 0 1 0 0 0 0 0 0 0 0 G 0 0 0
  10  FREQ           VARIABLE            1 1 0 0 0 0 1 0 0 0 0 0 0 0 0
  11  F3             VARIABLE            0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
  12  F2             VARIABLE            0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
  13  F1             VARIABLE            0 1 0 0 0 0 0 0 0 0 0 C 0 0 0
  14  IN2            VARIABLE            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  15  IN1            VARIABLE            0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                               SEQUENCE OF PROCESSING FOR TEST  SAVE _FREQ_
ORDER
VECT  ORDER   RANK         NAME           TYPE                          TEXT
INDEX VECTOR
  1     4      0        $S_W0004       ASSERTION          F1 =  10000
                                                          TARGET:  F1;

  2     5      0        $S_W0005       ASSERTION          F2 =  100
                                                          TARGET:  F2;

  3     6      0        $S_W0006       ASSERTION          F3 =  10
                                                          TARGET:  F3;

  4     8      0        V              VARIABLE           LOCAL

  5    10      0        FREQ           VARIABLE           LOCAL

  6    14      0        IN2            VARIABLE           LOCAL

  7    15      0        IN1            VARIABLE           LOCAL

  8    11      1        F3             VARIABLE           LOCAL

  9    12      1        F2             VARIABLE           LOCAL

 10    13      1        F1             VARIABLE           LOCAL

 11     2      2        $S_W0002       ASSERTION          IF FREQ<F1&FREQ>=F2 THEN RANGE =  1
                                                          ELSE      IF FREQ<F2&FREQ>F3 THEN RANGE =  2
                                                          ELSE RANGE =  0
                                                          TARGET:  RANGE
                                                          SOURCE:  FREQ,  F3,  F2,  F1;

 12     9      3        RANGE          VARIABLE           LOCAL

 13     1      4        $S_W0001       CONJUNCTION        IF RANGE>0 THEN (<IN1, IN2>  = FREQ_GEN(FREQ,V,RANGE))
                                                          SOURCE:  V,  RANGE,  FREQ;

 14     3      4        $S_W0003       ASSERTION          RANGE >  0
                                                          SOURCE:  RANGE;
```

A.4 FILE INPUT-OUTPUT

This example illustrates the updating of an inventory file, named INV, based on transactions contained in a sequential file, called TRANS. A record in TRANS has two fields: KEY and an array A. A record in INV is found corresponding to the KEY in each of the records in TRANS and is updated based on the sum of the corresponding array A. General functions ACCESS and IACCESS are used to read, and SAVE and ISAVE to store, records from SAM and ISAM files respectively.

Corresponding sequence reports and Equate-atlas code are given after the specification.

```
     /* NOPAL TEST SPECIFICATION SOURCE INPUT, FILE: SAPLIST */

NOPAL PROCESSOR OPTIONS SPECIFIED: SAPLIST,XREF1,CODE,SEQ,NOXREF2,SOURCE2,TRACE=3,DEBUG=ALL


STMT NO.
--------

    1        NOPAL SPEC IO;
    2           DCL 1 TRANS: FILE,
    2                  2 R: RECORD ARRAY(*),
    2                     3 KEY : INTEGER,
    2                     3 A: REAL ARRAY(20);
    3           DCL 1 OLD_INV: FILE,
    3                  2 OLD_IR: RECORD ARRAY(*),
    3                     3 OLD_B: REAL,
    3                     3 OLD_C: REAL,
    3                     3 OLD_D: REAL;
    4           DCL 1 NEW_INV: FILE,
    4                  2 NEW_IR: RECORD ARRAY(*),
    4                     3 NEW_B: REAL,
    4                     3 NEW_C: REAL,
    4                     3 NEW_D: REAL;
    5           TEST PROCESSTRANS;
    6              MEAS;
    7                 ASRT:  I = SUBS('NEW_B,NEW_C,NEW_D,PTR_NEW_IR,PTR_OLD_IR,KEY,A,SUMVAL,
    7           OLD_B,OLD_C,OLD_D,K,NEW_IR,OLD_IR,R',*)
    7                          TARGET: I;
    8                 ASRT:  J = SUBS('A:2',10)  TARGET:J;

                       /* READ RECORD K */
    9                 ASRT:  SUMVAL(I) = SUM(A(I,J),J)  TARGET: SUMVAL(I)
    9                      SOURCE: A(I,J);
   10                 ASRT: K(I) = KEY(I)    TARGET: K(I)
   10                      SOURCE: KEY(I);

                       /* UPDATE RECORD IR */
   11                 ASRT:  PTR_OLD_IR(I) = KEY(I)  TARGET:PTR_OLD_IR(I)
   11                      SOURCE: KEY(I);
   12                 ASRT:  NEW_B(I) = OLD_B(I) + SUMVAL(I)  TARGET: NEW_B(I)
   12                      SOURCE: SUMVAL(I),OLD_B(I);
   13                 ASRT:  NEW_C(I) = OLD_C(I)  TARGET: NEW_C(I)
   13                      SOURCE: OLD_C(I);
   14                 ASRT:  NEW_D(I) = OLD_D(I)  TARGET: NEW_D(I)
   14                      SOURCE: OLD_D(I);
   15                 ASRT:  PTR_NEW_IR(I) = KEY(I)  TARGET: PTR_NEW_IR(I)
   15                      SOURCE: KEY(I);

   16              LOGIC: [DUM;

   17           DIAG DUM: PRINT = MSG;
   18           MESSAGE MSG: ' NOTHING ';
   19           FUNC SUBS,TYPE=E;
   20           FUNC SUM,TYPE=E;
   21        END IO;
```

```
                         WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION IO                              PAGE   1

                                                                   1  1  1  1  1  1
                                     1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
                                    ---------------------------------------------
   1   PROCESSTRANS         TEST     | 0 12  0  0  0  0  1  1  0  1  0  1  0  1  1
   2   DUM                  DIAGNOSIS| 0  0  0  0  0  0  0  0  0  5  0  0  0  0  0
   3   SUBS                 VARIABLE | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   4   SUM                  FUNCTION | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   5   A                    VARIABLE | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   6   KEY                  VARIABLE | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   7   K                    VARIABLE | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   8   PTR_OLD_IR           VARIABLE | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
   9   OLD_B                VARIABLE | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  10   NEW_B                VARIABLE | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                         WEIGHTED ADJACENCY MATRIX FOR NOPAL SPECIFICATION IO                              PAGE   1

                                                                   1  1  1  1  1  1
                                     1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
                                    ---------------------------------------------
  11   OLD_C                VARIABLE | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  12   NEW_C                VARIABLE | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  13   OLD_D                VARIABLE | 1  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  14   NEW_D                VARIABLE | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
  15   PTR_NEW_IR           VARIABLE | 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
                          SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION IO                                PAGE   1

ORDER
VECTOR  ORDER    RANK          TYPE                         NAME
INDEX   VECTOR
   1       3       0    ITERATION VARIABLE           SUBS
   2       4       0    EVAL OR CONTROL FUNCTION     SUM
   3       5       0    GLOBAL VARIABLE              A
   4       6       0    GLOBAL VARIABLE              KEY
   5       9       0    GLOBAL VARIABLE              OLD_B
   6      11       0    GLOBAL VARIABLE              OLD_C
   7      13       0    GLOBAL VARIABLE              OLD_D
   8       1       1    TEST                         PROCESSTRANS
   9       2       2    DIAGNOSIS                    DUM
  10       7       2    GLOBAL VARIABLE              K
                          SEQUENCE OF PROCESSING FOR NOPAL SPECIFICATION IO                                PAGE   1

ORDER
VECTOR  ORDER    RANK          TYPE                         NAME
INDEX   VECTOR
  11       8       2    GLOBAL VARIABLE              PTR_OLD_IR
  12      10       2    GLOBAL VARIABLE              NEW_B
  13      12       2    GLOBAL VARIABLE              NEW_C
  14      14       2    GLOBAL VARIABLE              NEW_D
```

```
   15        15        2    GLOBAL VARIABLE           PTR_NEW_IR


                              INTRA TEST SEQUENCING PROCESSTRANS
                              ANALYSIS OF THE ADJACENCY MATRIX
                                                    1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
                                  1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
                                  - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  1  SM_W0001      ASSERTION      0 0 0 0 0 0 0 0 017 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  2  SM_W0002      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  3  SM_W0003      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  4  SM_W0004      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  5  SM_W0005      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
  6  SM_W0006      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
  7  SM_W0007      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
  8  SM_W0008      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
  9  SM_W0009      ASSERTION      0 0 0 0 0 0 0 0 017 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
 10  DUM           DIAGNOSES      0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 11  SUBS          VARIABLE       1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 12  I             VARIABLE       0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0
 13  J             VARIABLE       0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 14  SUM           VARIABLE       0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 15  A             VARIABLE       0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 16  SUMVAL        VARIABLE       0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 17  KEY           VARIABLE       0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 18  K             VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 19  PTR_OLD_IR    VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 020 0 0 0
 20  OLD_B         VARIABLE       0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 21  NEW_B         VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0
 22  OLD_C         VARIABLE       0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 23  NEW_C         VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0
 24  OLD_D         VARIABLE       0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 25  NEW_D         VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0
 26  PTR_NEW_IR    VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 020 0
 27  R             VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 019 019 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 28  TRANS         VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0 0 0 0 0
 29  OLD_IR        VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 019 019 0 0 0 0 0 0 0 0
 30  OLD_INV       VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019 0 0 0
 31  NEW_IR        VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 019
 32  NEW_INV       VARIABLE       0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                              SEQUENCE OF PROCESSING FOR TEST PROCESSTRANS
ORDER
VECT  ORDER    RANK          NAME           TYPE                 TEXT
INDEX VECTOR
  1     11      0           SUBS          VARIABLE        GLOBAL / SOURCE /


  2     14      0           SUM           VARIABLE        GLOBAL / SOURCE /


  3     28      0           TRANS         VARIABLE        GLOBAL / SOURCE /
 4    30     0    OLD_INV        VARIABLE        GLOBAL / SOURCE /

 5     1     1    SM_W0001       ASSERTION       I
                                                  =  SUBS('NEW_B,NEW_C,NEW_D,PTR_NEW_IR,PTR_OLD_IR,KEY,A,SUMVAL,     OLD
                                                 TARGET:  I
                                                 SOURCE:  SUBS;

 6     2     1    SM_W0002       ASSERTION       J =  SUBS('A:2',10)
                                                 TARGET:  J
                                                 SOURCE:  SUBS;

 7    12     2    I              VARIABLE        LOCAL

 8    13     2    J              VARIABLE        LOCAL

LOOP-1 STARTS: SUBSCRIPT I ITERATES FROM 1 TO *
 9    27     3    R              SUBSCRIPT_VA    GLOBAL / SOURCE /

10    17     4    KEY            SUBSCRIPT_VA    GLOBAL / SOURCE /

11     4     5    SM_W0004       ASSERTION       K(I) =  KEY(I)
                                                 TARGET:  K
                                                 SOURCE:  I,  KEY;

12     5     5    SM_W0005       ASSERTION       PTR_OLD_IR(I) =  KEY(I)
                                                 TARGET:  PTR_OLD_IR
                                                 SOURCE:  I,  KEY;

13     9     5    SM_W0009       ASSERTION       PTR_NEW_IR(I) =  KEY(I)
                                                 TARGET:  PTR_NEW_IR
                                                 SOURCE:  I,  KEY;

14    18     6    K              SUBSCRIPT_VA    GLOBAL / TARGET /

15    19     6    PTR_OLD_IR     SUBSCRIPT_VA    GLOBAL / TARGET /

16    26     6    PTR_NEW_IR     SUBSCRIPT_VA    GLOBAL / TARGET /

17    29     7    OLD_IR         SUBSCRIPT_VA    GLOBAL / SOURCE /
```

```
18    20     6        OLD_B          SUBSCRIPT_VA     GLOBAL / SOURCE /

19    22     6        OLD_C          SUBSCRIPT_VA     GLOBAL / SOURCE /

20    24     6        OLD_D          SUBSCRIPT_VA     GLOBAL / SOURCE /

21     7     9        SM_W0007       ASSERTION        NEW_C(I) =  OLD_C(I)
                                                      TARGET:  NEW_C
                                                      SOURCE:  I,  OLD_C;

22     8     9        SM_W0008       ASSERTION        NEW_D(I) =  OLD_D(I)
                                                      TARGET:  NEW_D
                                                      SOURCE:  I,  OLD_D;

23    23    10        NEW_C          SUBSCRIPT_VA     GLOBAL / TARGET /

24    25    10        NEW_D          SUBSCRIPT_VA     GLOBAL / TARGET /

25    15    11        A              SUBSCRIPT_VA     GLOBAL / SOURCE /

 LOOP-2 STARTS: SUBSCRIPT J ITERATES FROM 1 TO     10
26     3     2        SM_W0003       ASSERTION        SUMVAL(I) =  SUM(A(I,J),J)
                                                      TARGET:  SUMVAL
                                                      SOURCE:  I,  J,  SUM,  A;
 LOOP-2 ENDS;

27    16     3        SUMVAL         SUBSCRIPT_VA     LOCAL

28     6     4        SM_W0006       ASSERTION        NEW_B(I) =  OLD_B(I)+SUMVAL(I)
                                                      TARGET:  NEW_B
                                                      SOURCE:  I,  SUMVAL,  OLD_B;

29    10     5        DUM            DIAGNOSES        PRINT = MSG;

30    21     5        NEW_B          SUBSCRIPT_VA     GLOBAL / TARGET /

31    31     6        NEW_IR         SUBSCRIPT_VA     GLOBAL / SOURCE /

32    32     7        NEW_INV        VARIABLE         GLOBAL / SOURCE /
 LOOP-1 ENDS;


LOOP SUMMARY TABLE :
  LOOP-1     FIRST NODE IS     9      LAST NODE IS    32      SUBSCRIPT IS I
  LOOP-2     FIRST NODE IS    26      LAST NODE IS    26      SUBSCRIPT IS J
```

```
           BEGIN EULATE PRCGRAM 'IO' $
C             NAMES                    INDEX
C               *** TEST MODULES ***
C               'PROCESSTRANS'      --  1
C
C               *** DIAGNOSES ***
C               'DUM'               --  1
C       $
C       DECLARATIONS OF SYSTEM VARIABLES  $
           DECLARE DIGITAL, LIST,'SYS.DIAG-FLAG'(1) $
           DECLARE DECIMAL, 'SYS.S-TIME' $
           DECLARE DECIMAL, 'SYS.D-TIME' $
           DECLARE DECIMAL, 'SYS.DUMMY' $
           DECLARE DECIMAL, 'SYS.NAME' $
           DECLARE DECIMAL, LIST,'SYS.#TESTS IN CONJ'(1) $
           DECLARE DIGITAL, LIST, 'SYS.TEST-FLAG'(1) $
           DECLARE DIGITAL, 'SYS.FLAG' $
           DECLARE DIGITAL, 'SYS.ASRT-FLAG' $
           DECLARE DECIMAL, 'SYS.TIM', 'SYS.TIME' $
           DECLARE DECIMAL, LIST, 'SYS.CLOCK'(6)  $
           DECLARE DECIMAL, 'SYS.I' $
           DECLARE DIGITAL, 'SYS.Y/N' $
           DECLARE DIGITAL, 'SYS.STATE' $
           DECLARE DIGITAL, 'SYS.SELECT' $
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' , 1 $
           DEFINE 'SYS.#TESTS' , 1 $
           DEFINE 'SYS.TRUE' , B'1' $
           DEFINE 'SYS.FALSE' , B'0' $
C       UUT POINT DEFINITIONS     $
C       MACRO DEFINITIONS $
           DEFINE 'PRT.TIME', 'SYS.CLOCK'(1), " ##:",
                  'SYS.CLOCK'(2), "##:", 'SYS.CLOCK'(3), "##" $
C       DECLARATIONS FOR USER DEFINED GLOBAL VARIABLES   $
           DECLARE DECIMAL, LIST, 'A' (2,10) $
           DECLARE DECIMAL, LIST, 'KEY' (2) $
           DECLARE DECIMAL, LIST, 'K' (2) $
           DECLARE DECIMAL, LIST, 'PTR-OLD-IR' (2) $
           DECLARE DECIMAL, LIST, 'OLD-B' (2) $
           DECLARE DECIMAL, LIST, 'NEW-B' (2) $
```

```
            DECLARE DECIMAL, LIST, 'OLD-C' (2) $
            DECLARE DECIMAL, LIST, 'NEW-C' (2) $
            DECLARE DECIMAL, LIST, 'OLD-B' (2) $
            DECLARE DECIMAL, LIST, 'NEW-D' (2) $
            DECLARE DECIMAL, LIST, 'PTR-NEW-IR' (2) $
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, 'I.1'  $
            DECLARE DECIMAL, 'J.1'  $
            DECLARE DECIMAL, LIST,'SUMVAL.1' (2) $
            DECLARE DECIMAL, 'SYS.DEC.01' $
            DECLARE DIGITAL, 'END-I.1' $
C $
C     USER DEFINED ATE FUNCTIONS $
C $
            INCLUDE "UPFLIB" $
C*********************************************************************$
C*********************************************************************$
C     DIAGNOSES PRCCS $
C*********************************************************************$
C*********************************************************************$
 1L00       DEFINE PROCEDURE, 'DUM' $
            RECORD " NOTHING " $
            'SYS.DIAG-FLAG'(1) = 'SYS.SELECTED' $
            END 'DUM' $
C*********************************************************************$
C*********************************************************************$
C     TEST PROCS $
C*********************************************************************$
C*********************************************************************$
 1100        DEFINE PROCEDURE, 'PROCESSTRANS' $
             'SYS.FLAG' = 'SYS.TRUE' $
             'SYS.ASRT-FLAG' = 'SYS.TRUE' $
             PERFORM 'GET.TIME ' $
             'SYS.S-TIME' = 'SYS.TIME' $
             RECORD "*** AFTER STEP 1100 *** ENTERED TEST PROCEDURE ",
                 ",'PROCESSTRANS' AT ",'PRT.TIME' $
C***WAVEFORM S/M-W0001 FOLLOWS: ********************$
C  NO CODE FOR SUBSCRIPTS & PARENT DCL. $
C***WAVEFORM S/M-W0002 FOLLOWS: ********************$
C  NO CODE FOR SUBSCRIPTS & PARENT DCL. $
             'END-I.1' = 'SYS.TRUE' $
             'SYS.VIRTITER' = 'SYS.FALSE'  $
             FOR 'S..I.1' = 1 THRU 16001 BY 1 THEN $
                 'SYS.VIRTITER' = NOT('SYS.VIRTITER') $
                 COMPARE 'SYS.VIRTITER' EQ, 'SYS.TRUE'  $
                 GOTC STEP  1105      IF NOGO $
                 'I.1' = 1  $
                 GOTC STEP  1110       $
 1105            'I.1' = 2  $
 1110           'SYS.ASRT-FLAG' = 'SYS.TRUE' $
C ISSUE AN ACCESS COMMAND.          27 $
                'ACCESS.PRM99' = C2 $
                PERFORM 'ACCESS' $
                'KEY' = ''ACCESS.PRM01' $
                'A' = ''ACCESS.PRM02' $
C***WAVEFORM S/M-W0004 FOLLOWS: ***********************$
                RECORD "*** AFTER STEP 1110 BEGIN ASSERTION S/M-W0004 "$
                'K'('I.1') = 'KEY'('I.1') $
C***WAVEFORM S/M-W0005 FOLLOWS: ***********************$
                RECORD "*** AFTER STEP 1110 BEGIN ASSERTION S/M-W0005 "$
                'PTR-OLD-IR'('I.1') = 'KEY'('I.1') $
C***WAVEFORM S/M-W0009 FOLLOWS: ***********************$
                RECORD "*** AFTER STEP 1110 BEGIN ASSERTION S/M-W0009 "$
                'PTR-NEW-IR'('I.1') = 'KEY'('I.1') $
C ISSUE AN ACCESS COMMAND.          29 $
                'IACCESS.PRMC1' = 'PTR-OLD-IR' $
                'IACCESS.PRM99' = 04 $
                PERFORM 'IACCESS' $
                'OLD_B' = 'IACCESS.PRM02' $
                'OLD_C' = 'IACCESS.PRM03' $
                'OLD_D' = 'IACCESS.PRM04' $
C***WAVEFORM S/M-W0007 FOLLOWS: ***********************$
                RECORD "*** AFTER STEP 1110 BEGIN ASSERTION S/M-W0007 "$
                'NEW-C'('I.1') = 'OLD-C'('I.1') $
C***WAVEFORM S/M-W0008 FOLLOWS: ***********************$
                RECORD "*** AFTER STEP 111C BEGIN ASSERTION S/M-W0008 "$
                'NEW-D'('I.1') = 'OLD-D'('I.1') $
C***WAVEFORM S/M-W0003 FOLLOWS: ***********************$
                RECORD "*** AFTER STEP 1110 BEGIN ASSERTION S/M-W0003 "$
C   FUNCTION CALL ON 'SUP' $
                'SUP.PRM01' = 'A'('I.1','J.1') $
                'SUP.PRM02' = 'J.1' $
 1115           PERFORM 'SUM' $
                'SYS.DEC.01' = 'SUM.RES' $
C   END OF FUNCTICN CALL ON 'SUM' $
                COMPARE 'SUMVAL.1'('I.1'),  EQ 'SYS.DEC.01' $
                RECORD "*** AFTER STEP 1115 END ASSERTION S/M-W0003 "$
                GOTC STEP  1120      IF GO $
                'SYS.FLAG' = 'SYS.FALSE' $
                'SYS.ASRT-FLAG' = 'SYS.FALSE' $
                RECCRD "*** AFTER STEP 1115 ASSERTION S/M-W0C03 EVALUATED TO ",
                   "FALSE" $
C***WAVEFORM S/M-W0006 FCLLOWS: ***********************$
 1120           RECCRD "*** AFTER STEP 1115 BEGIN ASSERTION S/M-W0006 "$
                'NEW-B'('I.1') = ('OLD-B'('I.1')+ 'SUMVAL.1'('I.1')) $
 1125           COMFARE 'SYS.ASRT-FLAG', EQ 'SYS.TRUE' $
                GOTC STEP  1130      IF NOGO $
                PERFORM 'DUM'  $
C ISSUE A SAVE COPMAND.          31 $
 1130           'ISAVE.PRMC1' = 'PTR-NEW-IR' $
                'ISAVE.PRMC2' = 'NEW_B' $
                'ISAVE.PRMC3' = 'NEW_C' $
                'ISAVE.PRMC4' = 'NEW_D' $
```

```
                    'ISAVE.PRM59' = 04 $
                    PERFORM 'ISAVE' $
                    COMPARE 'EAD-I.1', EQ  'SYS.TRUE' $
                    GOTO STEP  1135      IF NOGO $
               END FOR $
 1135          'SYS.TEST-FLAG'(1) = 'SYS.TESTED' $
               PERFORM 'GET.TIME' $
               RECORD "*** AFTER STEP 1135 RETURNING FROM TEST PROCEDURE ",
                  "","'PROCESSTRANS' AT ",'PRT.TIME' $
               END 'PROCESSTRANS' $
C*****************************************************************$
C*****************************************************************$
C     SYSTEM VARIAELE INITIALIZATION AND FIRST ENTRY POINT    $
C*****************************************************************$
C*****************************************************************$
E  1200         PERFORM 'GET.TIME'  $
               RECORD "!X TESTING UUT: IO" $
               RECORD  'SYS.CLOCK'(4), "DATE ##/",'SYS.CLOCK'(5), "##/" ,
                           'SYS.CLOCK'(6), "at TIME", 'PRT.TIME' $
               'SYS.TIP' = '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 'PROCESSTRANS' $
 1300          'SYS.FLAG' = 'SYS.TRUE' $
               PERFORM 'PROCESSTRANS' $
C $
 1400          PERFORM 'GET.TIME' $
               RECORD "FINISHED 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) $
               RECORD "DURATION " , 'PRT.TIME' $
               REMOVE ALL  $
 1405          RECORD "DO YOU WISH TO RERUN THIS PROGRAM? (Y/N)" $
               WAIT-FOR MANUAL-DATA-GO-NOGO $
               GOTO STEP 1410 IF NOGO $
               RECORD "!P" $
               GOTO STEP  1200    $
 1410          RECORD "TERMINATE EQUATE PROGRAM 'IO' !P" $
 1415          FINISH $
               TERMINATE EQUATE PROGRAM 'IO' $
```

Index

References

1. Ashcroft, E.A. and W.W. Wadge. Lucid, A Non-procedural Language with Iteration. *CACM 20*, 7 (July 1977).

2. Atkinson, R.R., B.H. Liskov and R.W. Scheifler. Aspects of Implementing CLU. ACM Annual Conference, Washington, D.C., Dec., 1978, pp. 123-129.

3. Balzer, R. et.al. Domain Independent Automatic Programming. Tech. Rep. RR-77-14, ISI, University of Southern California, Oct., 1974.

4. Biermann, A.W. Approaches to Automatic Programming. In *Advances in Computers, Vol. 15*, M. Rubinoff and M. Yovits, Eds., Academic Press, 1976.

5. Brinch-Hansen, P. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977.

6. Chamberlin, D.D. The 'Single-Assignment' Approach to Parallel Processing. FJCC, 1971.

7. Chang, Y. *Automatic Test Program Generation*. Ph.D. Th., The Moore School, University of Pennsylvania, 1977.

8. Dahl, O.J., B. Myhrhaug and K. Nygaard. The SIMULA 67 Common Base Language. Publication S-22, Norwegian Computing Center, Oslo, 1970.

9. Darringer, John A. and Mark S. Laventhal. A Study of the Use of Abstractions. Research Report RC7184, IBM T.J. Watson Research Center, June, 1978.

10. Dewar, R.K., A. Grand, S.C. Liu and J.T. Schwartz. Program by Refinement, as Exemplified by the SETL Representational Sub-language. *TOPLAS, ACM 1*, 1 (July 1979).

11. Earley, J. Towards an Underatanding of Data Structures. *Comm. ACM 14*, 10 (Oct. 1973), 617-627.

12. Friedman, D.P. and D.S. Wise. CONS Should Not Evaluate Its Arguments. Tech. Rep. 44, Computer Science Dept., Indiana University, Bloomington, 1975.

13. Gana, J.L. *An Automatic Program Generator for Model Building in Social and Engineering Science*. Ph.D. Th., The Moore School, University of Pennsylvania, 1978.

14. Goguen, J.A., J.W. Thatcher and E.G. Wright. An Initial Algebra Approach to the Specification, Correctness, and implementation of Abstract Data Types. In *Current Trends in Programming Methodology, Vol. 4*, R.T. Yeh, Ed., Prentice Hall, 1978.

15. Graubert, R. A Case Study of Using a Non-Procedural Language for Automatic Testing of Electronic Equipment. Master Th., The Moore School, University of Pennsylvania, 1979.

16. Green, C. The Design of PSI Program Synthesis. Second International Conference on Software Engineering, San Francisco, Oct., 1972.

17. Gries, D. and N. Gehani. Some Ideas on Data Types in High Level Languages. *CACM 20*, 6 (June 1977).

18. Guttag, J. V., E. Horowitz and D.R. Musser. The Design of Data Type Specifications. In *Current Trends in Programming Methodology*, R.T. Yeh, Ed., Prentice-Hall, 1978.

19. Guttag, J. V., E. Horowitz and D.R. Musser. Abstract Data Types and Software Validation. *CACM 21*, 12 (Dec. 1979).

20. Guttag, John V. Abstract Data Types and the Development of Data Structures. *CACM 2*, 6 (June 1977).

21. Hammer, Michael, W. Gerry Howe, Vincent J. Kruskal and Irving Wladawsky. A Very High Level Programming Language for Data Processing Applications. *Comm. ACM 20*, 11 (November 1977).

22. Heidorn, George E. Automatic Programming Through Natural Language Dialogue: A Survey. Research Report RC6074, IBM T.J. Watson Research Center, December, 1975.

23. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. *CACM 12*, 10 (Oct. 1969), 576-583.

24. Hoare, C.A.R. Proof of Correctness of Data Representations. *Acta Informatica 1* (1972).

25. Hoffman, C.M. Design and Correctness of a Compiler for a Non-procedural Language. *Acta Informatica* 9 (1978).

26. Homer, E.D. An Algorithm for Selecting and Sequencing Statements as a Basis for Problem Oriented Programming Languages. Proceedings of the ACM National Meeting, 1966.

27. Iverson, K.E. *A Programming Language*. Wiley, New York, 1962.

28. Jensen, K. and N. Wirth. *Pascal - User Manual and Report*. Springer-Verlag, 1974.

29. Kennedy, K. and J.T. Schwartz. An Introduction to the Set Theoretic Language SETL. *Comp. & Math. with Appl. 1* (1975).

30. Kessels, J.L.W. A Conceptual Framework for a Non-procedural Programming Language. *CACM 20*, 12 (Dec. 1977).

31. Knuth, D.E. *The Art of Computer Programming. Vol. 1: Fundamental Algorithms.* Addison-Wesley, 1969.

32. Kowalski, R.A. Algorithms = Logic + Control. *Comm. ACM 22*, 7 (July 1979), 424-436.

33. Leavenworth, B.M. Non-procedural Data Processing. *The Computer Journal 20*, 1 (1977).

34. Liskov, B.H., A. Snyder, R. Atkinson and C. Schaffert. Abstraction Mechanisms in CLU. *CACM 20*, 8 (Aug. 1977).

35. Liskov, B.H. and S. Zilles. Specification Techniques for Data Abstractions. *IEEE Trans. on Software Engineering 1*, 1 (March 1975).

36. Lu, K.S. Program Optimization Based on a Non-procedural Specification. Proposal for Ph.D. research. University of Pennsylvania, April 1980.

37. Manna, Z. *Mathematical Theory of Computation.* McGraw Hill, 1974.

38. Parnas, D.L. A Technique for the Specification of Software Modules with Examples. *CACM 15*, 5 (May 1972).

39. Pnueli, A., K.S. Lu and N.S. Prywes. Model Program Generator: System and Programming Documentation. The Moore School, University of Pennsylvania, Fall, 1980.

40. Pnueli, A. Scheduling an Equational Specificaion. Unpublished memo. 1979

41. Prywes, N.S., C. Tinaztepe and Y.K. Chang. Automatic Test Program Generation. Autotestcon, IEEE, Nov., 1977.

42. Prywes, N.S., A. Pnueli and S. Shastry. Use of a Non-procedural Specification Language and Associated Program Generator in Software Development. *TOPLAS 1*, 2 (Oct. 1979).

43. Ramirez, J.A. *Automatic Generation of Data Conversion Programs Using a Data Description Language.* Ph.D. Th., The Moore School, University of Pennsylvania, 1973.

44. Rin, N.A. *Automatic Generation of Business Data Processing Programs from a Non-Procedural Language.* Ph.D. Th., The Moore School, University of Pennsylvania, 1976.

45. Ruth, G.R. Protosystem I: An Automatic Programming System Prototype. Technical Memo TM-72, MIT Laboratory for Computer Science, July, 1976.

46. Sangal, Rajeev. The Nopal Program Generator: System and Programming Documentation, Vol. 1, 2 and 3. Tech. Rep. 80, The Moore School, University of Pennsylvania, March, 1980.

47. Schlesinger, S. and L. Sashkin. POSE: a Language for Posing Problems to a Computer. *CACM 10*, 5 (May 1967).

48. Shastry, S.K., A. Pnueli and N. Prywes. Non-Procedural Computer Programming with Model. Proc. of First Int. Computer Software and Application Conf., 1977.

49. Shastry, S., A. Pnueli and N.S. Prywes. Basic Algorithms Used by the MODEL System for Design of Programs. The Moore School, University of Pennsylvania, Feb., 1979.

50. Shastry, S.K. *Verification and Correction of Non-procedural Specifications in Automatic Generation of Programs*. Ph.D. Th., The Moore School, University of Pennsylvania, May 1978.

51. Shaw, Mary, W.A. Wulf and R.L. London. Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators. *Comm. ACM 20*, 8 (Aug. 1977), 553-564.

52. Tesler, L.G. and H.J. Enia. A Language Design for Concurrent Processes. SJCC, 1968.

53. Thatcher, J.W., E.G. Wagner and J.B. Wright. Specification of Abstract Data Types Using Conditional Axioms. Research Report RC6214, IBM T.J. Watson Research Center, Sept., 1976.

54. Tinaztepe, C. and N.S. Prywes. Generation of Software for Computer Controlled Test Equipment for Testing Analog Circuits. *IEEE Trans. on Circuits and Systems* (June 1979). Special issue on automatic analog fault diagnosis.

55. Tinaztepe, C., R. Sangal, H. Che and N.S. Prywes. The Nopal Program Generator: System and Programming Documeniation. The Moore School, University of Pennsylvania, 1979.

56. Tinaztepe, C., R. Sangal and N.S. Prywes. Automatic Generation of Atlas Programs for Computer Controlled Test Equipment. Autotestcon, IEEE, Nov., 1978.

57. Weisman, C. *LISP 1.5 Primer*. Dickenson Publishing Company, Inc., Belmont, Calif., 1967.

58. Wirth, N. *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood, N.J., 1976.

59. Wirth, N. Modula: a Language for Modular Multiprogramming. *Software Practice and Experience 7* (1977).

60. Wirth, N. The Use of Modula. *Software Practice and Experience 7* (1977).

61. Zloof, M.M. Query-by-example. Proc. NCC, AFIPS, 1975.

62. Zloof, M.M. and S.P. de Jong. The System for Business Automation (SBA): Programming Language. *Comm. ACM 20*, 6 (June 1977).

DAT
ILM