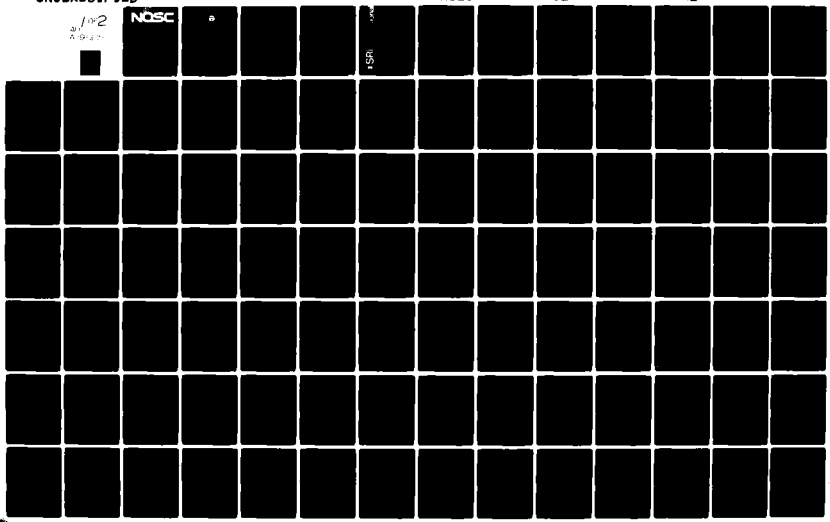
AD-A091 271 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK. VOLUME--ETC(U)
JUN 79 W L SUTTON, B A SILVERBERG, L ROBINSON N00123-76-C-0195
UNCLASSIFIED NOSC-TD-366-VOL-2 NL



A091270

LEVEL III

(12)

NOSC

NOSC TD 366

NOSC TD 366

DTIC
ELECTE
NOV 7 1980
C

Technical Document 366

AD A091271

THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK

Volume II: The Languages and Tools of HDM

June 1979

Prepared for

Naval Ocean Systems Center

DDC FILE COPY

Approved for public release; distribution unlimited

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

80 11 04 043

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

SL GUILLE, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

The HDM Project was funded under Navy Element 62721N. The work leading to this publication was the result of NOSC Contract N00123-76-C-0195 with SRI International. The principal researchers were Dr. Karl Levitt (Volume III), Mr. Lawrence Robinson (Volume I), and Dr. Brad A. Silverberg (Volume II). The Navy Technical Monitor for the work performed under contract was W. Linwood Sutton, NOSC Code 8324.

Reviewed by
J. B. Balistrieri, Acting Head
C^3I Facilities Engineering &
Development Division

Under authority of
V. J. Monteleon, Acting Head
Command, Control, Communications
and Intelligence Systems Department

(18) NOSC (19) TR-366-Vol-2

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
NOSC Technical Document 366

2. GOVT ACCESSION NO.
AD-A091

3. RECIPIENT'S CATALOG NUMBER
271

4. TITLE (and Subtitle)
(6) THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK
Volume II: The Languages and Tools of HDM

5. TYPE OF REPORT & PERIOD COVERED
(9) Technical document

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)
(10) W. Linwood Sutton (contract monitor)
Brad A. Underwood

8. CONTRACT OR GRANT NUMBER(s)
(15) N00123-76-C-0195

9. PERFORMING ORGANIZATION NAME AND ADDRESS
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025
Lawrence Robinson
Karl N. Levitt

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
62721N

11. CONTROLLING OFFICE NAME AND ADDRESS
(11) Naval Ocean Systems Center
San Diego, CA 92152

12. REPORT DATE
June 1979

13. NUMBER OF PAGES
106

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)
Naval Ocean Systems Center, Code 832
San Diego, CA 92152
(12) 114

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
abstract machines, abstraction, formal specification, hierarchical structure, Hierarchical Development Methodology (HDM), modules, software development process, software development tools, software methodology, Specification and Assertion Language (SPECIAL).

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
HDM (the SRI Hierarchical Development Methodology) is an approach to software development that attempts to structure the overall development process by providing a unified framework that addresses most aspects of system development. Volume II of the HDM Handbook provides a self-contained description of the languages of HDM and the principal online tools. The primary language of HDM is the module specification language SPECIAL (Specification and Assertion Language). SPECIAL embodies the concepts of HDM described in Volume I.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410281

SRI International

THE HDM HANDBOOK

Volume II: The Languages and Tools of HDM

Deliverable A006

SRI Project 4828
Contract N00123-76-C-0195

June 1979

By: Brad A. Silverberg, Computer Scientist
Lawrence Robinson, Computer Scientist
Karl N. Levitt, Program Manager

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Naval Ocean Systems Center
San Diego, California 95152

Attention: W. Linwood Sutton, Contract Monitor

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A

CONTENTS

I Introduction 3

II Specification Languages 5

A. Definition of the System Requirements 6

B. Formal Specifications 6

C. Types of Formal Specifications 8

D. Aspects of SPECIAL 9

III Overview of Abstract Machine and Module Specification 13

IV The Expression Level of SPECIAL 17

A. Expressions 17

B. Types 17

1. Primitive Types 18
2. Subtypes 19
3. Constructed Types 20
4. The Role of UNDEFINED 20
5. Type Specifications 21

C. Extended BNF Notation 22

D. Operations on Types 22

1. Predefined Types 22
2. Sets 23
3. Vectors 26
4. Structures 27
5. Union Types 27

E. Miscellaneous Forms 28

1. Quantified Expressions 28
2. Characterization Expressions 28
3. Conditional Expressions 30

V The Specification Level of SPECIAL: Module Specifications 31

A. V-functions, C-functions, and OV-functions 31

B. Module Specifications 31

C. Declarations 33

D. TYPES Paragraph 33

E. PARAMETERS Paragraph 35

F. DEFINITIONS Paragraph 36

G. EXTERNALREFS Paragraph 37

H. ASSERTIONS Paragraph 39

I. FUNCTIONS Paragraph 40

1. V-function Definition 41
2. O- and OV-function Definitions 44

VI The Specification Level of SPECIAL: Mapping Functions 49

A. Paragraphs of a Mapping Function Specification 50

B. A Small Mapping Function Example 54

C. Mapped Specifications 56

VII The Hierarchy Specification Language 57

VIII ILPL Specifications 59

A. Qualities of an Implementation Language for HDM 59

B. Overview of ILPL 61

C. The Expression Level of ILPL 62

D. The Specification Level of ILPL 63

1. The "Informational" ILPL Paragraphs 63
2. ILPL Programs 65

E. Argument Passing in ILPL 72

IX The Tools of HDM 75

A. General Presentation of the System 75

B. The Module Handler 76

C. The Mapping Function Handler 78

D. The Implementations Checker 79

E. The Interface Handler 80

F. The Hierarchy Handler 81

G. Command Abbreviations 82

A. Mapped Specifications for a Bounded Stack 87

B. Grammars for SPECIAL and ILPL 91

Foreword

Work on the languages and tools of HDM involved many people over a number of years. The person most involved with this effort was Lawrence Robinson. HDM, as a comprehensive development methodology, is primarily due to his efforts. Olivier Roubine was heavily involved in the language design efforts and in the design and implementation of the tools. Other significant contributions were made by Karl Levitt, Bob Boyer, Brad Silverberg, and Bernard Mont-Reynaud.

I Introduction

In this volume we present the languages and tools of the SRI Hierarchical Development Methodology (HDM). The languages provide a way of recording and communicating decisions made throughout the stages of system design, specification, implementation, and verification. The tools assist the system developer during this development process. The current set of tools is used primarily to determine whether certain well-formedness and consistency criteria are satisfied.

The languages of HDM are intended to capture the concepts and computational model described in Volume I. SPECIAL (Specification and Assertion Language) is used to specify modules and mapping functions. HSL (Hierarchy Specification Language) is used to describe the structuring of modules into machines, and machines into systems. ILPL (Intermediate Level Programming Language) is used to record module implementation decisions. In addition, the final implementation code is written in some executable programming language such as Pascal, Euclid, Ada, etc. Such implementation languages could also be considered "languages of HDM", though we take a narrower view and restrict our attention to SPECIAL, HSL, and ILPL.

In the next chapter of this volume we discuss the general need for specification languages, and describe how our particular specification language (SPECIAL) relates to HDM. Chapter III gives an overview of module specification. Chapter IV presents the expression level of SPECIAL, while Chapters V and VI respectively present specifications for modules and mapping functions. Chapter VII describes HSL, and Chapter VIII describes ILPL. Finally, Chapter IX discusses the tools of HDM.

II Specification Languages

In the last few years much research has been devoted to specification methods. The objective of a specification method is to describe the external behavior of a program without describing or constraining its internal implementation. To describe the external behavior of a program, a specification must provide the following information:

- what operations it can be asked to do,
- what information it must be given,
- what results are obtained, including error indications, and
- the effects of prior operations on subsequent operations.

One might also wish to include information about the resource requirements of the program, but such performance specifications are still beyond the ability of existing specification methods. Thus, we restrict our attention to <u>functional</u> specifications.

The specification method should be able to define completely the external behavior of the programs, since programs that have unintended or incompletely understood side-effects can inflict nasty surprises. It is also very desirable that the specification of a program not constrain the subsequent implementation of the program, provided that the external behavior is as specified. Any implementation that possesses the properties demanded by the specification is said to be "correct".

Our demand that a specification be precise and unambiguous dictates the use of specification languages with well-defined syntax and semantics, and rules out the use of informal languages such as English.

Some of the benefits of a formal specification language are:

- The presence of a precise specification provides a rugged interface between program units in a complex system. The programs that implement a specification can be designed and implemented, and subsequently modified, without consideration of the use that is to made of them. Similarly, the programs that are to invoke the specified facilities can be designed and implemented without concern for any specific details of

the implementation. The specification completely defines the interactions between the programs and hides everything else.

- The rigor of a formal specification ensures a more complete design, and a better statement and understanding of details. Important decisions cannot be overlooked or equivocated about, and the decisions are all documented unambiguously.

- The specification can serve as an interface between designer and implementor, as well as between implementor and user.

- The specification provides a reference for the documentation of the program, for the construction of tests, for the maintenance and development of the program, and for re-implementation of the program.

- The use of a formal specification provides the opportunity for program proof and eventually perhaps even automatic programming.

A. Definition of the System Requirements

A requirement definition must express precisely what the user wants from the system, in contrast to a (functional) specification which states exactly what the system is to do. The distinction is somewhat fuzzy, though in typical usage, the user's requirements are much less specific. He needs a certain effect achieved, but may not need to predetermine the precise command or sequence of commands to be used, nor may he be concerned with precisely what formats are used or what the error indications will be. Occasionally a user will require exact compatibility, and then his requirement definition will have to be a specification. Otherwise, the broader the requirement definition can be made while still requiring the function needed by the user, the more freedom is left to the designers to optimize the system and the better they will be able to make the system.

B. Formal Specifications

The necessary mathematical basis for formal requirements definition is at this time regretfully not well understood, and no acceptable general approach to requirements currently exists. On the other hand,

methods for formal <u>specifications</u> do exist. The purpose of a specification is to provide a precise definition of the intended behavior of a system of programs, a program, or program component. This definition should not constrain the means by which the implementation achieves the desired results, for instance by presuming certain internal data representations or algorithms. But the specification should completely define the externally observable behavior of the program, so that the only effects that can be discovered by a user are precisely defined by the specification. In effect, the specification defines a "black box" of known behavior and unknown internal construction.

A specification should be complete, in that for any operation, its result can be deduced from the specification; the specification should also be consistent, in that it not contain contradictory (hence, unimplementable) constraints, such as "X=0 AND X=1".

Since a formal specification is a mathematical object, one can demonstrate in a formal, rigorous manner that a given specification possesses certain postulated properties. For example, we may want to prove from a stack specification that "Top(Push(x)) = x". The prospect of powerful mechanical theorem provers that can establish complex properties of specifications automatically makes this aspect particularly attractive.

Coupled with a modular approach to system design, the formal specification method significantly reduces the severity of program modification and maintenance problems. Information about a given module's behavior is communicated to users solely through the module's specification, and not through its program code. Thus, suppose that a given module's specification remains constant while its implementation is changed (and the implementation stays consistent with the specification); then the modules using the given module are not affected and do not have to be changed.

Traditional informal specifications can still serve a useful purpose. They can be used to convey the meaning of a module in a more accessible though necessarily less rigorous manner. Informal notations may also serve as instructions or suggestions to the implementor,

something especially desirable since formal specifications are meant to be implementation-independent. HDM does not provide guidelines for such informal specifications, but does permit informal comments to be embedded into a specification.

C. Types of Formal Specifications

For the purpose of this exposition, we distinguish among three approaches to module specifications: operational, state-machine, and axiomatic. The HDM approach falls under the state-machine category.

An operational specification is a set of programs that by definition exhibits the desired behavior. For the purposes of specification, simplicity and clarity rather than efficiency are the most important considerations. Though in a strict sense an operational specification may be viewed as a grossly inefficient (though by definition correct) implementation, we typically do not include the specification itself when we consider "implementations". Rather, an operational specification serves as a reference point against which (the functional behavior of) other implementations are compared. The problem of proving the correctness of an implementation thus becomes the problem of proving program equivalence: demonstrating that the specification and implementation programs are computationally equivalent.

An operational specification language is usually chosen for its elegance and clean semantics. Examples of such languages are pure Lisp, the theory of Boyer-Moore [1], and the axiomatized subset of Pascal [4].

A number of problems result from the operational approach. Sometimes we may want to leave some cases unspecified (e.g., the order of evaluation of operands in arithmetic expressions), or allow some latitude in the exact output produced as long as it satisfies some output constraint (e.g., the solution to some equation within a given tolerance). This is not possible with an operational specification; the specification <u>overspecifies</u> the problem. An operational specification can also misleadingly suggest a strategy for implementation. It is misleading because (functional) specifications are constructed without

regard for efficiency; this is not true for implementations. In addition, there is the hazard that an operational specification, as a program, may not always terminate. For these reasons, we do not consider operational specifications further.

The concepts of the state-machine approach have been fully explained in Volume I. The essential notions are:

- An explicit state is referred to and modified.

- Non-procedural mathematical expressions are used to characterize state changes and returned values.

Of the axiomatic approaches, the algebraic approach of John Guttag [3] has attracted the most interest. This approach consists of contraints, called axioms, on particular sequences of operations such that the effect for an arbitrary sequence of operations can be deduced from these axioms. In contrast to the state-machine approach, the algebraic approach involves only an implicit concept of state. The approach sometimes leads to shorter specifications and produces specifications in a form consistent with some verification systems. However, we favor the state-machine approach for the following reasons:

- It gives the effect of a module operation in an explicit manner, rather than indirectly through a set of axioms. Our experience is that the explicit approach is more intuitive and transparent, and specifications are more easily developed and comprehended. Algebraic specifications have been criticized for their opaqueness, and for their failure to provide "feel" for what in being specified.

- It has been applied to the specification of some important large systems, such as SIFT, an ultra-reliable avionics computer with software-implemented fault tolerance [7], and PSOS, a provably secure operating system [5].

D. Aspects of SPECIAL

Though SPECIAL is not a programming language in the conventional sense, it has greatly benefited from the lessons learned in programming language design. The basis for these advances has been a better understanding of the programming process and an elucidation of the underlying mathematics.

SPECIAL provides a means for recording decisions for modules and mapping functions developed while using HDM. Specifically, SPECIAL provides ways to: (1) specify a module's internal data structures and constraints upon the initial values of those data structures; (2) characterize a module's operations, their effects on the internal data structures of the machine, and the conditions under which the operations may be invoked; (3) protect a module's objects; and (4) support the interrelationships among the modules of a machine and between machines at adjacent levels.

These considerations have dictated the high level structure of SPECIAL. Below are some of the issues we considered important.

A specification language should be:

Clear and Descriptive: As we discussed in the critique of the operational approach, clarity and simplicity are of utmost importance. Specifications are meant to be read and understood, not executed. A good specification language should enable the designer to manipulate abstract entities as he distills the essential characteristics and omits the procedural details. The specification language need not contain features common to executable programming languages, features whose only purpose is to speed-up the compilation process or to assist the compiler produce efficient code. As a result, we have made SPECIAL a declarative, non-procedural language, rather than an imperative, procedural one.

Semantically Well-Defined: The need to place specification languages on firm mathematical ground, and the derived benefits, are discussed at length in the previous section. SPECIAL is based on first-order predicate calculus and set theory, which also serve as the basis for several verification systems.

At the present time, only a subset of SPECIAL has been formalized. Using the theory of Boyer-Moore, we have given formal semantics to most, but not all, of the essential HDM concepts [2]. The major goal of the continuing HDM effort is to formalize the entirety of our specification language. Until that effort is complete, however, when we speak of SPECIAL being "formal", we are referring to the formalized subset.

Well-Supported by a Comprehensive Set of Tools: A comprehensive set of tools can aid in detecting inconsistencies. There are two general types of inconsistencies in a specification: semantic and syntactic. A specification with a semantic inconsistency is one that contains some constraint no implementation can satisfy, e.g., the constraint "x=0 AND x=1". While it is not possible to write a program that can detect all such semantic errors, many can be exposed by mechanical theorem provers. On the other hand, syntax errors can always be detected. The current set of HDM tools is primarily concerned with detecting such errors of syntax. (At SRI we use the powerful Boyer-Moore theorem prover to help detect the more difficult semantic errors.) Syntax errors fall into two general classes: context-free and context-sensitive errors. The context-free class relates to the specification's phrase structure and is easily handled by well-known algorithms for context-free parsing. Context-sensitive errors usually reflect naming inconsistencies (e.g., two distinct formal parameters with the same name to a given operation or function) or data type violations (e.g., the expression

b = 17 + TRUE

where b is of type BOOLEAN). Strong typing in SPECIAL is part of the language, and specifications that violate the strong typing rules are regarded as being "ill-formed".

We have tried to keep SPECIAL free of "innovations" and "new features". The real ideas of SPECIAL are to be found in the concepts of HDM, as discussed in Volume I, and not in the details of the language design. SPECIAL is just a medium, and the success of SPECIAL as a language design can be judged by how well it captures HDM concepts without introducing extraneous details.

III Overview of Abstract Machine and Module Specification

An abstract machine specification is a definition of the machine's external properties. Since each module of a given machine is only loosely dependent -- if at all -- on the other modules in the machine, we will (for simplicity) concentrate in this chapter on the specification of one-module abstract machines. The detailed example of Volume III contains several instances of nontrivial multi-module machines.

A module is conveniently viewed as containing internal data structures and providing operations to users of its interface. The data structures define the <u>state</u> of a module. (As we present module specifications more completely, we will see there are additional components to the state.) The operations provide the means by which the state may be accessed and/or modified from the outside.

A module's data structures are characterized by <u>state-functions</u>, which are to be distinguished from mathematical functions. There are two types of state-functions: primitive and derived. The primitive ones are the only ones that actually define state. A derived state-function is some expression over the primitive state-functions; as such, it is simply a shorthand abbreviation. The specification must give each state-function's functionality (i.e., its domain and range); for the primitive state-functions, the specification must also specify the constraints upon the initial values; for the derived ones, it must give the derivation.

The execution of an operation may either "raise an exception" or "return normally". In general, a normal return involves a returned value and a state change. The returned value is described in terms of constraints it must satisfy. The state change is described by relating the post-invocation values of the state-functions to their pre-invocation values.

An exception return occurs when one of the <u>exception conditions</u> associated with the operation is satisfied (raised). An exception condition depends on the state of the machine and on the values of the

operation's actual arguments. An exception return includes an indication of which exception was raised. In the current version of SPECIAL, no state change may occur if an exception is raised. For the specification of a bounded stack "push" operation, a typical exception condition would be an expression that detects an attempt to push a value onto a full stack.

For a given sequence of operations, the final value of a state-function is determined inductively: the specification for the state-function gives its initial value, and the specifications for the operations derive post-invocation state-function values from their pre-invocation values.[1]

We next present a small example to illustrate these points. A few of the terms used below have not been explained, but the reader should still be able to follow the example. The example defines an unoounded push-down stack of objects called stack_elems. The data structures are:

```
VFUN Access(INTEGER i) -> stack_elem s;
     HIDDEN;
     INITIALLY  FORALL INTEGER i: Access(i) = UNDEFINED;

VFUN Size() -> INTEGER i;
     HIDDEN;
     INITIALLY  i = 0;
```

The data structure Access defines the current elements in the stack, and Size gives the current stack size. Initially the stack is empty; hence, Access(i) is UNDEFINED for all INTEGER i, and Size() is 0.

The operations are:

---

[1]To be completely precise, we should say that we determine the final <u>constraint</u> on the state-function's value, rather than the value itself. This is because both the initial value and the operation effects are specified through constraints which in general may not be uniquely satisfied.

```
VFUN Top() -> stack_elem s;
     EXCEPTIONS  empty: Size()=0;
     DERIVATION  Access(Size());

OFUN Push(stack_elem s);
     EFFECTS
         'Size() = Size()+1;
         Access(Size()+1) = s;

OVFUN Pop() -> stack_elem s;
     EXCEPTIONS  empty: Size()=0;
     EFFECTS
         'Size() = Size()-1;
         s = Access(Size());
         'Access(Size()) = UNDEFINED;
```

(In the above, 'f denotes the post-invocation value of the state-function f. We call this the state-function's quoted value.) If a state-function value is not mentioned in an operation's effects section, it is assumed that the operation does not change the given state-function's value.

Briefly, Top causes no state change and returns the top of the stack as long as the stack is not empty. Push places a new element on the stack by increasing the stack size by one, leaving all previous elements where they were, and inserting the new element on the top. Note that Push causes a state change though it neither returns a value nor has any exceptions. Finally, the stack is popped with the operation Pop. Pop shrinks the size of the stack by one, returns the pre-invocation top as its value, and specifies that the previous top now "no longer exists". (This will be explained in more detail in later chapters.)

There is another formulation we could give that would make Size a derived, rather than a primitive, state-function. This would be to define Size as the cardinality of the set {INTEGER i | Access(i) ≠ UNDEFINED}. As a result, two effects would be removed: the one that increments Size() in Push, and the one that decrements Size() in Pop.

IV The Expression Level of SPECIAL

SPECIAL can be decomposed into two distinct language levels, the expression level and the specification level. The expression level is concerned with the way expressions are constructed; the specification level is concerned with the way expressions are put together into module specifications and mapping functions. The expression level of SPECIAL is very much like the expression level of other modern languages. Readers familiar with languages such as Pascal, Alphard, CLU, etc., will find few new ideas here.

In this chapter we discuss expressions, types, operations on types, and miscellaneous expression forms.

A. Expressions

The concept of expressions is at the heart of SPECIAL. Expressions are composed into assertions (predicates) that in turn are used to describe: initial conditions for primitive state-functions; derivations for derived state-functions; exception conditions, state-transformations, and returned values for operations; mapping functions; and invariant properties.

An expression denotes a value. At the base level, an expression is a constant or a variable. A constant refers to a single, unique value. A variable assumes its value from a specified set of values; in any given instance, the variable denotes one of those values.

More complicated expressions are constructed by composing an operator with a sequence of operands (arguments). Each argument is itself an expression. Operators may be predefined (e.g., + or NOT) or user-defined. User-defined operators may be state-functions, value-returning operations, or user-defined expression forms.

B. Types

A type is a set of values. Values of a given type may be

manipulated only through the set of operations defined on that type. With the exception of the special value UNDEFINED as explained below, every constant is of a particular, distinct primitive type. A variable's type is the range of values it may assume. An operator has a type associated with each argument and with its result. Thus, the type of an expression is: if the expression is elementary (i.e., a constant or variable), then the type is the type of the given constant or variable; otherwise, the type is the result type of the expression's operator.

The semantics of SPECIAL specify the type for each constant and for the arguments and result of each predefined operator. Correspondingly, SPECIAL provides mechanisms by which variables and user-defined formal arguments and results are declared, i.e., associated with a type.

SPECIAL requires that types of actual arguments be "compatible" with the types of the formal arguments. These restrictions are imposed to prevent the writing of some "meaningless" expressions.

There are three classes of types in SPECIAL: primitive types, subtypes, and constructed types.

1. Primitive Types

Primitive types are the building blocks from which other types are constructed. As such, primitive types cannot be decomposed. The only value that two primitive types may (and do) have in common is the distinguished value UNDEFINED, which is an implicit member of every type.

SPECIAL has three primitive types: predefined, scalar, and designator. The predefined types are the familiar INTEGER, REAL, BOOLEAN, and CHAR. The usual kinds of operations apply to these types. In SPECIAL, however, these types possess their true <u>mathematical</u> properties. Hence, the notions of overflow, round-off error, etc., which apply to the machine counterparts of these types, are not applicable to these types in SPECIAL.

The BOOLEAN constants are TRUE and FALSE. CHAR constants are denoted by single characters within single quotes, e.g., 'A'.

Scalar types are user-defined (unordered) finite sets of enumerated symbolic constants. For example, the scalar type primary_color could be the set {red, blue, yellow}. The only built-in operation permitted on scalar types is equality. There is no added generality in including scalar types in SPECIAL since the values of a scalar type can always be represented by integers, for example. However, scalar types do serve useful purposes. First, they improve readability. In a program that deals with the days of the week, Monday, Tuesday, ..., Sunday are much more mnemonic than 1,2,...,7. Second, since scalar types have only a limited set of built-in operations, type-checking can invalidate such nonsense expressions as "red + yellow < blue", which would be legal with integers.

Designator types are the means by which abstract data types are defined in SPECIAL. For example, if we wanted a stack data type, we would declare "stack" as a designator type. Each value of a designator type, called a <u>designator</u>, is used to name an object of an abstract data type. There is a one-to-one correspondence between designators and conceptualized abstract objects of a given type. Thus, each designator value of our type "stack" denotes the name of a distinct stack object. The only built-in operations that apply directly to designators are equality and NEW. (The designated object itself may be modified by supplying the designator for that object as an argument to a state-changing operation. In this way, designators are somewhat analogous to pointers in programming languages.) The operation NEW takes the name of a designator type as its argument and returns a never-used designator for that type. There are no designator constants except UNDEFINED; after initialization, the only way to create a designator is through the use of NEW.

The set of all currently allocated designators constitutes part of the state of the module that supports the given designator type. Thus, designator allocation is a state-changing operation.

2. Subtypes

A subtype is a specified subset of a given type. We will see how

to specify subtypes once we present the ways to specify sets. A subtype inherits the operations of its constituent type.

3. Constructed Types

There are three flavors of constructed types: union, aggregate, and structured. The value set of a union type is the union of the value sets of its several constituent types. A union type inherits the operations of its constituent types.

An aggregate type is a homogeneous collection of objects from a given constituent type. The two kinds of aggregate types are sets and vectors. A set data type's value set is the powerset of its constituent type's value set. For example, the type SET_OF {a,b} has the value set: { {}, {a}, {b}, {a,b} }. Vectors are like sets except they are (1) ordered and (2) always finite. Vectors in SPECIAL are really indexable sequences.

A structured type is similar to a PL/I structure or a Pascal record. An example is the type complex_number with two REAL components, realpart and imagpart.

The operations defined for the above constructed types are discussed at length in a later section.

4. The Role of UNDEFINED

It is often useful in a specification to indicate that a given object in a module does not exist. We use the particular symbol UNDEFINED (also, "?") to represent "no value" or non-existence. If we used a particular conventional value in each case -- say, 0 for integers -- to indicate "no value", the implementation would be constrained to supply that same value even though it may not "exist". "?" has the advantage that it does not constrain implementations. For example, if we pop a stack, the specification could either leave the old top value, or change it to "?". In the former case, the implementation would be forced to leave that value around even though it may be inaccessible by the module's operations. In the latter case, however, the implementation is not forced to represent this "non-existent" popped

value.

In a specification, "?" can only be meaningfully manipulated with the equality relationals (= and ~=). For all other appearances of "?" as an actual argument to a given function or operation, the result is "?". For example, both ?=? and ?~=2 are TRUE, while ?+2 is ?.

Because "?" symbolizes the lack of a value, it is never actually used in an implementation. Thus, no value-returning operation should be specified to return "?" for any feasible set of actual arguments. In addition, since a user of a module cannot pass "?" as an actual argument to an executing program, there is no need in a module specification to specify a returned value, a state change, or an exception condition for an UNDEFINED argument.

"?" is a distinct, implicit member of every SPECIAL type. Thus the type INTEGER is actually the value set

$$\{ \ldots, -2, -1, 0, 1, 2, \ldots \} \cup \{?\}$$

5. Type Specifications

Each variable, formal argument, and formal result must be <u>declared</u> (or associated with a type), before use. A type specification is either *an explicit type specification* or a *named type*. Examples of some explicit type specifications for the different kinds of types are:

predefined	INTEGER BOOLEAN
scalar	{red, blue, yellow}
subtype	{1, 3, 5, 7} {INTEGER i ¦ i > 0} {1 .. 10}
aggregate	SET_OF INTEGER VECTOR_OF {red, blue, yellow}
structured	STRUCT_OF(REAL realpart; REAL imagpart)
union	ONE_OF(INTEGER,REAL)

A named type associates an identifier with a type specification, e.g.,

complex_number: STRUCT_OF(REAL realpart; REAL imagpart)

Then a complex_number variable cx can be declared

```
complex_number cx
```

A designator type must be a named type. We write

```
stack: DESIGNATOR
```

to declare the stack designator type, and then

```
stack s
```

to declare the variable s of type stack.

C. Extended BNF Notation

The notation we use for syntax equations in this document is based on an extended form of BNF (Backus-Naur Form). In this notation, nonterminals are enclosed in angle brackets; the left side of a rule is separated from the right side by a colon; and special characters that are terminals appear within single quotes, e.g., ';'. We also use the following meta-constructs:

[...]*	the enclosed elements appear 0 or more times
[...]+	the enclosed elements appear 1 or more times
[...]?	the enclosed elements are optional
[...|...]	an alternative between the enclosed elements

In the above, [and] serve as meta-parentheses. When there is no danger of ambiguity, we may omit the meta-parentheses.

D. Operations on Types

1. Predefined Types

The built-in operations on predefined types are the logical, arithmetic and relational operators that apply to objects of types INTEGER, REAL, BOOLEAN, and CHAR. The logical operators are AND, OR, negation (NOT or ~), and implication (=>); each takes BOOLEAN argument(s) and has a BOOLEAN result. The arithmetic operators are +, - (unary or binary), *, /, and MOD. Each operates on numbers, which is an object of type INTEGER or REAL. Objects of both numeric types can be

freely intermixed as arguments to the arithmetic operators. An operator having only INTEGER arguments has an INTEGER result; otherwise the result is REAL. Each relational operator returns a BOOLEAN result. The ordering relationals <, <=, >, and >= each take two numbers as arguments. The equality relationals, = and ~=, permit objects of any two non-disjoint types as arguments.[1]

There are at the present time no built-in operators besides equality for objects of type CHAR.

2. Sets

SPECIAL provides the following set operators: UNION, INTER, DIFF, INSET, SUBSET, and CARDINALITY. They correspond respectively to: union, intersection, set difference, membership, set inclusion, and cardinality.

The UNION operator takes two set-valued arguments and has a set-valued result. Assume (without loss of generality) that the types of the two arguments are respectively of the form

```
ONE_OF(SET_OF A1,...,SET_OF An)
```

and

```
ONE_OF(SET_OF B1,...,SET_OF Bm),
```

where each Ai and Bj is a type specification. (Note that ONE_OF(SET_OF x) is equivalent to SET_OF x.) Then the type of the result is the "ONE_OF" of SET_OF ONE_OF(Ai,Bj) for 1<=i<=n, 1<=j<=m. For example, if s1 has the type SET_OF INTEGER and s2 has the type ONE_OF(SET_OF REAL, SET_OF CHAR), then the type of s1 UNION s2 is

```
ONE_OF(
        SET_OF ONE_OF( INTEGER,REAL ),
        SET_OF ONE_OF( INTEGER,CHAR ))
```

The intersection operator is defined in a similar way. It also takes two set-valued operands and has a set-valued result. We assume

---

[1]Since "?" is part of every type, two types can never, strictly speaking, be disjoint. For simplicity, however, we will regard the intersection of two types as being empty if the only element in common is "?".

the above "ONE_OF" form for the types of the arguments. The INTER operator requires that the arguments be sets whose constituent types are not disjoint. More precisely, we require[1]

$$(\cup A_i) \cap (\cup B_j) \neq \{\}. \qquad (1)$$

The result type of INTER is the "ONE_OF" of SET_OF xij, where xij is the intersection of Ai and Bj. For example, if s1 has type SET_OF ONE_OF(INTEGER,REAL) and s2 has type ONE_OF(SET_OF INTEGER, SET_OF REAL), then the type of the result of s1 INTER s2 is

ONE_OF(SET_OF INTEGER, SET_OF REAL),

the type of s2. If s3 has type ONE_OF(SET_OF INTEGER, SET_OF CHAR), the result type of s1 INTER s3 is SET_OF INTEGER.

The DIFF operation also takes two set-valued operands and has a set-valued result. The disjointness restriction (1) from above also applies to DIFF. The type of the result is the type of the first argument.

The INSET operation takes a first operand, say, with type A and a second set-valued operand, say with type of the above ONE_OF form. The restriction here is that

$$A \cap (\cup B_j) \neq \{\}$$

The result of INSET is BOOLEAN.

The SUBSET operation takes two set-valued operands and has a BOOLEAN result. It also has the disjointness restriction (1).

CARDINALITY allows any finite set as an argument and returns an INTEGER result.

In addition to the above conventional set operators, SPECIAL also provides MIN and MAX. MIN and MAX each take a finite set of numbers as argument and respectively return the set's minimum or maximum element. If the argument set is empty, the result is UNDEFINED.

Sets are specified with forms called set constructors. An <u>extensional constructor</u> requires an enumeration of the individual elements. The elements of an extensional constructor may in general be

[1] $\cup$ denotes set union, $\cap$ set intersection.

expressions, though they are typically constants. Examples of extensionally constructed sets are {red, blue, yellow} or SET(red, blue, yellow) for the set containing the primary colors, and {} or SET() for the empty set.

An <u>intensional constructor</u> supplies a necessary and sufficient property of the elements. The intensional constructor for sets has the form

'{' <typespec> <symbol> '|' <pred> '}'

where <pred> is a predicate. The value of such an expression is the set that contains all values in the type <typespec> such that when substituted for free occurrences of <symbol> within <pred>, <pred> is TRUE. We have already used an intensional constructor to specify the set of positive integers: {INTEGER i | i > 0}. The set of odd integers between 1 and 9 (inclusive) could be defined by the intensional constructor

{INTEGER i | 1<=i AND i<=9 AND (i MOD 2 = 1)}

or by the extensional constructor

{1, 3, 5, 7, 9}.

The <u>subrange set constructor</u> is used to specify sets of numbers. The constructor defines a set by giving expressions for the endpoints of a closed interval; the set contains all elements in the interval. The general form for a subrange constructor is:

'{' <expr> '..' <expr> '}'

If both endpoint expressions have integer type, the resulting set also has integer type; otherwise, it has real type. The first expression gives the left endpoint, the second the right endpoint.

A subrange constructor {e1 .. e2} with result type t is equivalent to the intensional constructor {t x | e1 <= x AND x <= e2}. For example, {1.0 .. 4.0} is equivalent to {REAL x | 1.0 <= x AND x <= 4.0}. Also, {1 .. 4} is equivalent to {INTEGER i | 1 <= i AND i <= 4}, whcih in turn is equivalent to {1, 2, 3, 4}. Note that {4 .. 3} denotes the empty set.

The intensional constructor is the most general one.

As we will see in later sections, there are constructors for vectors and structures, too.

3. Vectors

There are two operations on vectors. LENGTH takes a vector-valued argument and has an INTEGER result. The value of LENGTH(v) is the number of elements in the vector v. The extraction operation is written v[i], where v is a vector-valued expression and i is an integer-valued expression in the range 1 to LENGTH(v). The result is the ith element of v, and its type is the constituent type of v; e.g., if v has type *VECTOR_OF REAL*, v[i] *has type REAL*.

An extensional vector constructor with n elements defines a vector with respective elements in positions 1 to n. The intensional constructor has the syntax

```
VECTOR '(' FOR <symbol>
    FROM <expr> TO <expr> ':' <expr> ')'
```

We define the value of an intensional constructor of the general form

```
VECTOR(FOR i FROM f_expr TO t_expr: f(i))
```

in terms of a closed interval of integers whose respective left and right endpoints are given by *f_expr* and *t_expr*. The elements of this interval are denoted by the ordered sequence e1,...,en, where n = t_expr - f_expr + 1. Now, the above constructor describes a vector v such that v[i] = f(ei), for 1<=i<=n. If n<1 (i.e., f_expr > t_expr), the vector is empty.

For example, the extensional constructor

```
VECTOR(1, 3, 5, 7)
```

describes the same vector as

```
VECTOR(FOR i FROM 1 TO 4: 2*i - 1)
```

and as

```
VECTOR(FOR j FROM 0 TO 3: 2*j + 1)
```

SPECIAL provides a shorthand notation for constants of type VECTOR_OF CHAR, i.e., character strings. As an illustration, "ABC" is the same as VECTOR('A','B','C'). (Note that the double quotes in "ABC" are required; they are not meta-symbols here.)

4. Structures

Structures have an extractor operation and two varieties of extensional constructors. If "complex_number" is a named type that represents STRUCT_OF(REAL realpart; REAL imagpart), then the extractors for the type are x.realpart and x.imagpart, where x is an object of complex_number type. The type of an extraction is the type of the associated selector. For example, x.realpart and x.imagpart are both REAL.

The two varieties of constructors work by selector name and by position, respectively. The selector name constructor reflects a view of structures that is consistent with that embodied by the extractor operation. This view sees a structure as an unordered collection of components, each identified by a selector name. Thus, the constructor specifies a structure by associating a value with each selector name, and does not depend on the order of the components in the type specification.

The positional constructor views a structure as an ordered collection of components, and disregards selector names. It specifies elements by position, in a similar manner as the extensional vector constructor.

For example, STRUCT(realpart: 1.0,imagpart: 0.0), STRUCT(1.0, 0.0), and STRUCT(imagpart: 0.0,realpart: 1.0) all denote the same complex_number. They are <u>not</u> the same as STRUCT(0.0,1.0).

5. Union Types

Suppose we have a variable of type ONE_OF(t1,t2), and wish the result of an expression to be f(x) if x is of type t1 and g(x) if x is of type t2. This is done with the TYPECASE expression. We write:

```
TYPECASE x OF
    t1: f(x);
    t2: g(x);
END
```

The type labels (here, t1 and t2) must refer to disjoint types; the type of the object variable (here, x) must be the union of all the type

labels; and the type of the entire expression is the union of the types of the component expressions (here, f(x) and g(x)).

E. Miscellaneous Forms

1. Quantified Expressions

A quantified expression represents a formula in first-order logic. There are two quantifiers: universal and existential, denoted respectively by FORALL and EXISTS. The syntax for quantified expressions is

```
[FORALL|EXISTS] <qualifications> ':' <pred>
```

where <qualifications> is a sequence of one or more <qual>s separated by semicolons.

```
<qualifications>: <qual>  [';' <qual>]*
```

A <qual> is a <simpledec> followed by an optional domain restriction. A <simpledec> is a kind of declaration, and is explained in Section V.C.

```
<qual>: <simpledec> <dom_restriction>?
<dom_restriction>:  INSET <setexpression>
                  |  '|' <pred>
                  |  <relop> <expr>
```

The domain restriction within a <qual> adds no generality but improves readibility. For example, we may write

```
FORALL INTEGER x INSET s : p(x)
```

to mean, "for all integers x in set s, p(x) is TRUE". This is equivalent to

```
FORALL INTEGER x : (x INSET s) => p(x)
```

Similarly, we may write

```
EXISTS INTEGER x | q(x) : p(x)
```

for the equivalent

```
EXISTS INTEGER x : q(x) AND p(x).
```

2. Characterization Expressions

Sometimes we write expressions in which an object with a particular value or property is used repeatedly. As a result, we have provided two forms, called characterization expressions, in which a group of

variables is first characterized (bound to values) and then used within an expression. The value of the form is the value of the expression.

The first of these forms, the LET expression, has the following syntax

```
LET <quals1> IN <expr>
```

where <quals1> is like <qualifications> above, except that within each <quals1> qualification the domain restriction is now mandatory.

As an example, consider a table implemented as a function tbl of one integer argument. The table has keys stored in even positions and associated values in subsequent positions. We would like to calculate f(v) + g(v), where v is the value (of type t) corresponding to the key k in the table. This can be expressed with

```
LET t v | EXISTS INTEGER i >= 0 : even(i) AND tbl(i)=k
                    AND tbl(i+1)=v
      IN f(v) + g(v)
```

If a qualifying expression does not <u>uniquely</u> characterize a qualified variable, then the variable may assume any one of the satisfying values in the expression body (the expression following the IN). Thus, the LET expression may in general denote any one of a set of values. With regards to the above example, this means that if v1, v2, ... all satisfied the qualifying expression, the value of the LET expression would be any one of f(v1)+g(v1), f(v2)+g(v2), ... (but not f(vi)+g(vj)) for i≠j).

In addition, if a qualification expression is unsatisfiable, then its associated qualified variable is bound to "?" in the expression body.

A restricted form of the LET expression is called a SOME expression. It has the form

```
SOME <qual1>
```

where

```
<qual1>: <simpledec> <dom_restriction>
```

(Note that <qual1> is the qualification that applies for <quals1> in the LET expression.)

An expression such as SOME REAL x | p(x) may assume as value any REAL x such that p(x) is TRUE. It is equivalent to LET REAL x | p(x) IN x.

3. Conditional Expressions

The conditional expression has the form

IF <pred> THEN <expr1> ELSE <expr2>

If <pred> is TRUE, the value is the value of <expr1>; if FALSE, it is the value of <expr2>; otherwise (if <pred> is "?"), it is "?". The type of the result is the union of the respective types of <expr1> and <expr2>.

V The Specification Level of SPECIAL: Module Specifications

In the discussion of the specification level, we show how module specifications are constructed from expressions at the expression level. But first we must introduce some vocabulary.

A. V-functions, O-functions, and OV-functions

So far we have talked about modules in terms of operations and state-functions. For historical reasons, however, SPECIAL specifies a module in terms of so-called V-functions, O-functions, and OV-functions.

V-functions have two purposes: to define the state of the module, and to provide information about the state to the module's users. In terms of our former model, V-functions serve simultaneously as state-functions and as value-returning operations that reveal but not modify state. The "V" in V-function conveys the value-returning idea.

The dual role for V-functions leads to shorter specifications, since the function used to characterize a data structure may also be used as an operation to query the value of the data structure. It is possible, however, to specify that a given V-function serve solely as a state-function. Such V-functions, called hidden V-functions, may not be invoked by an executing implementation program. Hidden V-functions, though, may be externally referenced in a module specification. We give an example that illustrates this important distinction in Section V.I.2.

The state-changing operations of our former model are now called O- and OV-functions. The distinction between these two types of functions is that an OV-function both causes a state change and returns a value, whereas an O-function only causes a state change.

B. Module Specifications

A module specification in SPECIAL consists of six paragraphs, each of which is optional in a given specification. In its most general form, the top-level structure looks like:

```
MODULE <symbol>

    TYPES
        <types body>

    PARAMETERS
        <parameters body>

    DEFINITIONS
        <definitions body>

    EXTERNALREFS
        <externalrefs body>

    ASSERTIONS
        <assertions body>

    FUNCTIONS
        <functions body>

END_MODULE
```

where <symbol> is the name of the module.

The TYPES paragraph contains the declarations for all internal named types (including designator types) used in the module specification. The PARAMETERS paragraph contains the declarations for *symbolic constants* called <u>module parameters</u>. Parameters are similar to V-functions, except that their values cannot be changed. The DEFINITIONS paragraph contains the definitions for macro-like auxiliary function definitions. The EXTERNALREFS paragraph contains the declarations for objects of other modules that are externally referenced in the specification. These objects include designator and scalar types; V-, O-, and OV-functions; and parameters. The ASSERTIONS paragraph contains (1) assertions that are constraints on the module's parameters, and (2) invariants of the module that need to be proved from its specification. Finally, the FUNCTIONS paragraph contains the definitions for all V-, O-, and OV-functions of the module.

In the next sections we look at each of these paragraphs in detail and give the respective syntax equations. In Appendix B we give the complete LALR(1) grammars for SPECIAL and ILPL, as used by the specification parsers in the tools. The syntax equations presented here were designed with clarity in mind, so they differ from the ones in the

Appendix. The languages described here also differ slightly from those currently supported by the tools, though the intention is to bring the tools completely up-to-date in the near future.

We first describe in detail the pervasive notion of declaration.

C. Declarations

In general, a declaration associates a type specification with a list of symbols. It is useful to distinguish between a declaration that refers to a single symbol and one that refers to possibly multiple (>=1) symbols. We call the former a <simpledec>, the latter a <fulldec>. Note that all <simpledec>s are also <fulldec>s, but not vice versa.

The relevant syntax equations are:

<declaration>: <fulldec> | <simpledec>

<fulldec>: <typespec> <symbollist>

<simpledec>: <typespec> <symbol>

<symbollist>: <symbol> [',' <symbol>]*

<declaration list>: <declaration> [';' <declaration>]*

We discuss type specifications in detail in the next section.

The following <declaration list> illustrates the above equations.

```
INTEGER i;
REAL x,y,z;
VECTOR_OF(CHAR) string;
STRUCT_OF(REAL realpart,imagpart) ix,jx;
stack s;
{red, yellow, blue} color
```

D. TYPES Paragraph

The TYPES paragraph contains the declarations for all internal named types, including designator types. The paragraph body consists of a sequence of type declarations. Each type declaration associates a list of symbols with either a type specification or the keyword DESIGNATOR.

The corresponding syntax equations are:

```
<types paragraph>:  TYPES  [<typedecl> ';']+

<typedecl>:  <symbollist> ':' [DESIGNATOR | <typespec>]

<typespec>:  <symbol>
           | <subtype>
           | <scalar type>
           | <predefined type>
           | <structure type>
           | <united type>
           | <aggregate type>

<subtype>: <setexpression>

<scalar type>: <scalar_type setexpression>

<predefined type>:  INTEGER | REAL | BOOLEAN | CHAR

<structure type>:  STRUCT_OF '(' <declaration list> ')'

<united type>:  ONE_OF '(' <typespeclist> ')'

<typespeclist>:  <typespec>  [',' <typespec>]*

<aggregate type>:  [SET_OF | VECTOR_OF]  <typespec>
```

The following is an example <types paragraph>.

```
TYPES
    stack: DESIGNATOR;
    pos_int: {INTEGER i | i > 0};
    number: ONE_OF(INTEGER, REAL);
    complex_number: STRUCT_OF(number imagpart, realpart);
    char_string: VECTOR_OF CHAR;
    color: {red, yellow, blue};
    one_to_ten: {1 .. 10};
```

Of the types declared in a given module specification, only the designator and scalar named types are "exported"; that is, they are the only types that may be referenced by other modules.

The set expression in a subtype specification may contain parameter and V-function references. The latter case means that it is possible for the value set of a subtype to vary dynamically, since V-functions can vary over time with the module's state.

E. PARAMETERS Paragraph

As repositories for state information, primitive V-functions are subject to change in value due to invocations of O- or OV-functions. In some module specifications there may be primitive V-functions that never change value. These V-functions attain their value upon initialization and are never modified thereafter. Such non-varying V-functions are usually (but not necessarily) specified as module parameters. Like primitive V-functions, module parameters constitute components of state.

As is the case for V-function invocations, parameter invocations are not permitted to effect a state change. This implies that a parameter may not return a previously unallocated designator, for designator allocation involves a modification of the underlying module state. A fixed set of designators, however, may be allocated and bound to a parameter at initialization time. Objects of non-designator types, on the other hand, are regarded as "system constants" that do not contribute to a module's state. Thus, a parameter may always return one of these objects.

The default constraint on parameter values is the assertion TRUE; that is, any value will do. The default may be overridden by constraints in the ASSERTIONS section.

Because the INITIALLY clause in a V-function specification is also in general a predicate that may not be uniquely satisfied, parameters provide no added generality over V-functions. Nevertheless, parameters do serve a useful purpose. They correspond to the intuitive notion of an implementation-provided immutable object (value or function) and provide an easy way to specify a module that depends on such objects whose bindings are not known a priori. Since there is no way to modify a parameter, the proof of constancy is trivial.

Parameters are typically used to:

1. Define a resource limitation that is not known a priori. For example, a module that provides a bounded stack might have a parameter max_stack_size.

2. Provide certain "built-in" functions, for example "sin".

3. Provide a fixed collection of designators that is part of the module's initial state, i.e., the designators are allocated at module initialization time.

All parameters are visible; that is, they may be called externally by implementations of other modules.

A parameter declaration associates a type specification with a set of symbols. The type specification gives the result type for each parameter named in the associated set of symbols. The declarations for the formal arguments of a parametric function follow the function's name in the list of symbols. The syntax equations are:

```
<parameters paragraph>:  PARAMETERS  [<parmdecl> ';']+

<parmdecl>:  <typespec> <symbol> <formalargs>?
                 [ ',' <symbol> <formalargs>? ]*

<formalargs>:  '(' <declaration list>? ')'
```

A sample PARAMETERS paragraph is thus:

```
PARAMETERS
    INTEGER max_stack_size;
    REAL sin(REAL x), cos(REAL x), epsilon;
```

F. DEFINITIONS Paragraph

The definitional facility in SPECIAL provides the mechanism by which auxiliary (mathematical) functions may be defined for use within the module specification. In such a definition, the specification gives the name of the function, declarations for the formal arguments, the type of the result, and the function body. Constants may be defined in a similar manner, with the formal argument list omitted.

The function body is a SPECIAL expression. When evaluated, the expression is not permitted to cause a state-change. This means that O-, OV-, and quoted V-functions may not appear within the expression. The names that may appear are: the definition's formal arguments, unquoted V-functions, parameters, named types, the given function's name itself (for recursive definitions), and the names of other definitions.

Function definitions and applications are subject to all the normal type-checking rules. This means that the type of the function body must

be consistent with the declared result type. The result type, in turn, must be consistent in a given application with the expected type.

The syntax for the DEFINITIONS paragraph is:

```
<definitions paragraph>:  DEFINITIONS [<definition> ';']+

<definition>:  <typespec> <symbol> <formalargs>?
                    IS <expression>
```

An example DEFINITIONS paragraph is:

```
DEFINITIONS
    BOOLEAN proper_subset(set s1, s2) IS
        (s1 SUBSET s2)  AND  s1~=s2;

    INTEGER fact(INTEGER i) IS
        IF i<=0 THEN 1 ELSE i * fact(i-1);
```

G. EXTERNALREFS Paragraph

Here we declare all externally referenced objects: parameters, functions, and named types from other modules referenced in the current specification. (Remember that the only referencable named types are designator and scalar types.) Since these objects may include state-changing operations (O- and OV-functions) and quoted V-functions, we see that a given module can alter the state of other modules. A full discussion of how external referencing induces module dependencies is contained in Volume I. Recall that the "externalref" relation induces a partial ordering.

For each externally referenced module, we first give the module's name and then declarations for its referenced objects. We have already seen declarations for named types and parameters; we now see how functions are declared.

A function declaration gives the function name, its kind (VFUN, OFUN, or OVFUN), declarations for its formal arguments, and a declaration for the result type (for V- and OV-functions only). We sometimes call a function declaration a <u>function header</u>.

The syntax equations for the EXTERNALREFS paragraph are:

```
<externalrefs paragraph>: EXTERNALREFS <externalgroup>+

<externalgroup>: FROM <symbol> ':' [<externalref> ';']+

<externalref>: <exttypedecl> | <parmdecl>
             | <functiondecl>

<exttypedecl>: <symbollist> ':'
                    [DESIGNATOR | <scalar type>]

<functiondecl>: <vfunheader>
              | <ofunheader>
              | <ovfunheader>

<vfunheader>:  VFUN  <symbol> <formalargs>
                     '->' <resultargs>

<ovfunheader>: OVFUN <symbol> <formalargs>
                     '->' <resultargs>

<ofunheader>:  OFUN  <symbol> <formalargs>

<resultargs>: <declaration>
```

For example, suppose that a given module manipulates stacks and arrays of real numbers, provided respectively by the stack and array modules. The referenced stack objects are the stack designator type and the Create_stack, Push, and Pop stack operations. The referenced array objects are the array designator type and the Create_array, Elt, and Change array operations. Elt(arr,i) denotes arr[i]; Change(arr,i,x) causes arr to be changed so that Elt(arr,i) = x. Note that the stack and array modules may provide other objects that can be externally referenced; the ones listed above, however, are the only ones referenced by our given module. Assuming that no other modules are externally referenced by our module, its EXTERNALREFS paragraph might be:

```
    EXTERNALREFS

    FROM array_module:
array: DESIGNATOR;
OVFUN Create_array(INTEGER lbd,ubd) -> array a;
VFUN  Elt(array a; INTEGER i) -> REAL x;
OFUN  Change(array a; INTEGER i; REAL x);

    FROM stack_module:
stack: DESIGNATOR;
OVFUN Create_stack() -> stack s;
OFUN  Push(stack s; REAL x);
OVFUN Pop(stack s) -> REAL x;
```

H. ASSERTIONS Paragraph

As we have previously discussed, the ASSERTIONS paragraph contains two kinds of assertions:

1. Constraints on parameters. The implementation's initialization program must bind the parameters to values that satisfy these constraints.

2. Module invariants. These invariants must be proved from the module specification.

The syntax for the paragraph is simple:

```
<assertions paragraph>: ASSERTIONS [<expression> ';']+
```

We further require that the type of each expression be BOOLEAN.

For a bounded stack module specification, where the bound is given by the integer parameter max_stack_size, the ASSERTIONS section would likely require the bound to be positive. That is,

```
ASSERTIONS
    max_stack_size > 0;
```

A more involved case would be for a module that provides as a parameter a sorting function in REAL vectors. Assuming that our sorter module does nothing else than provide this sorting function, its entire specification might be:

```
MODULE sorter

    DEFINITIONS
INTEGER #occurrences(VECTOR_OF REAL v; REAL x) IS
        CARDINALITY({INTEGER i | v[i] = x});

    PARAMETERS
VECTOR_OF REAL  sort(VECTOR_OF REAL v);

    ASSERTIONS
FORALL VECTOR_OF REAL v:
    (LENGTH(sort(v)) = LENGTH(v))

              AND

    (FORALL i | 1 <= i AND i <= LENGTH(v):
        #occurrences(v,v[i]) = #occurrences(sort(v),v[i]))

              AND

    (FORALL i | 1 <= i AND i < LENGTH(v):
        sort(v)[i] <= sort(v)[i+1]);

END_MODULE
```

These assertions state that the sorted vector is a permutation of the argument vector, and that the elements of the sorted vector are indeed sorted. Thus, these assertions capture formally the intuitive notion of a sort function.

I. FUNCTIONS Paragraph

We finally get to what is usually the most important paragraph in a specification, the FUNCTIONS paragraph. This paragraph contains the definitions for all the module's V-, O-, and OV-functions.

There are three kinds of function definitions, one for each of the three kinds of functions (V-, O-, and OV-). The information contained in each kind of definition naturally varies with the kind of function being defined. One section all three in general have in common is the optional ASSERTIONS section. This section, using the same syntax as the ASSERTIONS paragraph, specifies a list of conditions that must be guaranteed by any program calling the given function. Most often these conditions depend on the function's arguments. The assertions are

regarded as assumptions that may be taken for granted ~~within the~~ whenever the function is invoked in an implementation. In implementation terms, this means that if a call is made upon a function when all of the function's assertions are not satisfied, then the function may possibly fail to execute properly.

As mentioned, the syntax for the ASSERTIONS section is the same as for the ASSERTIONS paragraph.

<assertions section>: <assertions paragraph>

1. V-function Definition

As we discussed in Section V.A, V-functions have two purposes: to define (part of) the state of the module, and to provide such state information to the module's users. Correspondingly, a V-function definition must define the V-function's behavior both as a state-function and as an operation.

A V-function definition starts off with the function's header. That is, it states that a V-function is being defined, and then gives the function's name, declarations for its formal arguments, and a declaration for its result.

The next part of the definition establishes whether the V-function may be referenced as an operation by programs outside the module, or whether it serves solely as a state-function. In the former case, we say the function is visible; in the latter, hidden.

If the function is hidden, the keyword HIDDEN appears here. Otherwise (if the function is visible), we characterize its exception conditions, if there are any. Note that since a hidden V-function may not be invoked in an implementation, it may not have an assertions section.

The following discussion of the exceptions section also applies to O- and OV-function definitions.

The exceptions section is a list of exception conditions, each typically of the form "exception_name: boolean_expression". If any of the exception condition expressions is TRUE when the function is called,

the function returns immediately with a notification of the raised condition's name. If (and only if) all exception condition expressions are FALSE does a normal return take place.

The order in which the exception conditions are listed is important: exception condition ex_i may be raised if and only if: (1) its associated expression is TRUE, and (2) the associated expressions for $ex_1,\ldots,ex_{i-1}$ are all FALSE. Once a satisfied exception condition is found, subsequent ones are not tested.

Sometimes there is insufficient information in a module to express conditions for the occurrence of an exception. This typically occurs when the implementation of a function requests resources of the module(s) it invokes. Since we may not be able to characterize such a condition explicitly, we use the special exception condition RESOURCE_ERROR to indicate that the invocation of the function could not be completed due to some exhaustion of resources at a lower level. (Note that this special exception condition is not of the typical "name: expr" form.) The program that implements the function in question is responsible for sorting out the causes of resource exhaustion and for returning an indication of RESOURCE_ERROR.

There is another exceptions condition not of the typical form: the EXCEPTIONS_OF construct. It has the form

EXCEPTIONS_OF <call> ';'

where <call> is an application of an external function. The interpretation of this construct is that for each exception condition of the named external function, a corresponding one in the current function is derived as follows: the exception name stays the same, and the expression is obtained by substituting the actual parameters of the <call> for the free occurrences of the external function's formal arguments in the original expression. The order of the exception conditions is preserved.

The EXCEPTIONS_OF construct is one of the two places where O- and OV-functions may be externally referenced in a specification. The other is in the EFFECTS_OF construct, which we discuss in the O-function definition subsection.

The syntax for the exceptions section is:

```
<exceptions>: EXCEPTIONS [<exception_condition> ';']+

<exception_condition>:  <symbol> ':' <expression>
                      | RESOURCE_ERROR
                      | EXCEPTIONS_OF <call>
```

We require that each exception condition <expression> be Boolean-valued.

If a visible V-function does not have any exception conditions, the exceptions section is omitted.

The next part of a V-function definition is the optional ASSERTIONS section.

The final part of the definition depends on whether the function is <u>primitive</u> or <u>derived</u>. Note that this is independent of whether it is hidden or visible.

If the function is primitive, this part of the definition characterizes the initial conditions for the function. The form is:

```
INITIALLY boolean_expression ;
```

We regard each (free) occurrence of the V-function's result variable in an INITIALLY expression as denoting the V-function's value for a given set of arguments, and universally quantify the INITIALLY expression over all possible argument sets. For example, in Chapter III we gave the V-function specification

```
VFUN Access(INTEGER i) -> stack_elem s;
     HIDDEN;
     INITIALLY  FORALL INTEGER i: Access(i) = UNDEFINED;
```

An equivalent INITIALLY expression is the more concise

```
INITIALLY  s = UNDEFINED;
```

The former INITIALLY expression can be obtained from the latter by substituting "Access(i)" for "s", and then quantifying over all INTEGER i.

If the function is derived, we provide its derivation. This form is

```
DERIVATION expr ;
```

subject to the constraint that the type of expr be compatible with the function's declared result type. The derivation is not a general

constraint on the V-function's value, but rather an expression that directly gives the V-function's value (similar to a DEFINITION).

The collected syntax for a V-function definition is thus:

```
<vfundefn>: VFUN <symbol> <formalargs> '->'
            <resultargs> ';'
            [ HIDDEN ';'  |  <exceptions>? ]
            <assertions section>?
            [ <initial cond>  |  <derivation> ]

<initial cond>: INITIALLY <expression> ';'

<derivation>: DERIVATION <expression> ';'
```

The following SPECIAL specifications are for the stack V-functions of Chapter III:

```
VFUN Access(INTEGER i) -> stack_elem s;
     HIDDEN;
     INITIALLY  s = ?;

VFUN Size() -> INTEGER i;
     HIDDEN;
     INITIALLY  i = 0;

VFUN Top() -> stack_elem s;
     EXCEPTIONS
         empty:  Size() = 0;
     DERIVATION
         Access(Size());
```

2. O- and OV-function Definitions

Since the definitions for O- and OV-functions are so similar, we present them together and draw the necessary distinctions when appropriate.

Two distinguishing characteristics of O- and OV-functions are:

1. They are visible.
2. They are permitted to effect a state-transformation.

The former dictates that we give an exceptions section, the latter that we define the function's state-transforming effects.

The first part of the definition gives the function's header. For O-functions, which do not return values, the form is

```
OFUN <symbol> <formalargs> ';'
```

The header for OV-functions includes a declaration for the result:

OVFUN <symbol> <formalargs> '->' <resultargs> ';'

The second part of the definition contains the exceptions section; if the function has no exceptions, it is omitted. A full discussion of the exceptions section is given in the foregoing discussion of V-function definitions.

The third part of the definition is the optional ASSERTIONS section. This section is treated in Section V.I.

The final part of the definition is the EFFECTS section. This required section defines the function's effect through a list of assertions that relate the post-invocation values of the module's V-functions to their pre-invocation values. A post-invocation value of a V-function is denoted by preceding the function's name with a single quote ('). All V-function values that do not appear quoted in the EFFECTS section of a given operation are left unchanged by that operation.

A few important points about the EFFECTS section must be emphasized.

1. Unlike exception conditions, effect assertions are unordered. The list of assertions is really a <u>conjunction</u> of assertions; the intended AND is replaced by a ";" for readibility purposes (although they <u>could</u> still be specified with explicit ANDs).

2. The assertions may possibly not uniquely determine the post-invocation values; that is, they are constraints, not assignments: mathematical equations that need to be satisfied, not program statements. For example, if f, g, and h are V-functions, the following list of effects

   ```
   'f() + 'g() = f() + g();
   'f() > f() + 1;
   'h() = (h() + 1) MOD 5;
   ```

   does not uniquely characterize f and g. It does however uniquely determine h.

3. The non-procedural nature of the EFFECTS section is illustrated again with the following example.

   ```
   'Size() = Size()+1;
   'Size() = Size()-1;
   ```

These are <u>not</u> assignments to Size; rather they are <u>inconsistent</u> constraints since 'Size() cannot equal Size()+1 and Size()-1 at the same time.

4. Assertions about quoted derived V-functions are really assertions about the quoted values of the underlying primitive V-functions.

5. If a V-function value is not mentioned in an EFFECTS section, it is assumed not to have changed.

6. If an OV-function is being defined, the EFFECTS section must also constrain the result.

O- and OV-functions of other modules can be referenced by means of the EFFECTS_OF construct. If "ofn" is the name of an external O-function with formal arguments f1,f2,..., then "EFFECTS_OF ofn(a1,a2,...)" denotes the list of effect assertions that would be obtained by taking ofn's entire effect list and substituting a1,a2,... for the free occurrences of f1,f2,... . For referenced OV-functions, the expression "x = EFFECTS_OF ovfn(a1,a2,...)" means that, "x is equal to the result of ovfn(a1,a2,...) and all effects of ovfn, with the appropriate argument substitutions, are TRUE".

The syntax for an O-function definition is thus:

```
<ofundefn>: OFUN <symbol> <formalargs> ';'
            <exceptions>?
            <assertions section>?
            <effects section>

<effects section>: EFFECTS [<effects expr> ';']+
```

where <effects expr> is a Boolean expression that may contain "EFFECTS_OF <call>" terms. Similarly, the syntax for an OV-function is:

```
<ovfundefn>: OVFUN <symbol> <formalargs>
                   '->' <resultarg> ';'
             <exceptions>?
             <assertions section>?
             <effects section>
```

The specifications for our stack operations Push and Pop are:

```
OFUN Push(stack_elem s);
     EFFECTS
        'Size() = Size() + 1;
        'Access(Size() + 1) = s;

OVFUN Pop() -> stack_elem s;
     EXCEPTIONS
        empty: Size() = 0;
     EFFECTS
        'Size() = Size() - 1;
        s = Access(Size());
        'Access(Size()) = ?;
```

Since the derivation for Top() is Access(Size()), the assertion "s = Access(Size())" is equivalent to the assertion "s = Top()".

An interesting example of how hidden V-functions may be used is the specification of a "Replace" stack operation that achieves its effect by externally modifying stack_module's state. The desired effect of

```
OVFUN Replace(stack_elem s1) -> stack_elem s2;
```

is

```
'Access(Size()) = s1  AND  s2 = Access(Size())
```

This effect would appear in Replace's EFFECTS section, and Access and Size would be declared in the EXTERNALREFS paragraph. Though hidden V-functions may be externally referenced in a module specification, they may not be externally called by an implementation program. In this case, Replace's implementation may <u>not</u> contain the sequence

```
s2 <- Access(Size());
Access(Size()) <- s1;
RETURN(s2);
```

Rather, it must use only the <u>visible</u> operations. The following achieves the desired effect:

```
s2 <- Pop();
Push(s1);
RETURN(s2);
```

Note that the EFFECTS section for Replace could not contain the effects

```
s2 = EFFECTS_OF Pop(); EFFECTS_OF Push(s1);
```

because these effects would produce the unsatisfiable constraint

```
'Size() = Size()-1  AND 'Size() = Size()+1
```

Thus, we see that the term "hidden" relates to implementation, not specification.

VI The Specification Level of SPECIAL: Mapping Functions

This chapter presents that part of SPECIAL that is used for writing mapping functions.

For each pair of adjacent abstract machines we must specify how the data structures of the upper machine are represented in terms of the data structures of the lower level. The mapping functions provide a medium for expressing such representation decisions. By employing SPECIAL for the mapping functions, the representations are expressed in a precise and concise notation.

An upper-level machine is represented in terms of a lower-level machine through one or more mapping functions. Each mapping function relates some subset of the upper-level modules to a subset of the lower-level modules. These subsets are called <u>submachines</u>. Each upper-level module must appear in one (and only one) mapping function; thus, if a module appears in several (contiguous) hierarchy levels, only the lowest-level occurrence is mapped. Two upper-level modules are in the same submachine if they share representation decisions. The "only if" of this statement, however, is not true: the partition need not be the smallest one; two modules may be grouped together in the absence of a shared representation decision purely for convenience or ease of specification. A full discussion of the ways upper-level modules can share representation decisions is contained in Volume I.

The basis for aggregation of lower-level modules is simply demand: the lower-level modules of a given mapping function are simply those that provide some part of the representation.

The collection of mapping functions for a given upper-lower machine pair is used to associate upper-level states with lower-level states. Each upper-level state is associated with a <u>set</u> of lower-level states. Distinct upper states must be represented by disjoint sets of lower states. The state of a machine, the reader will recall, is defined by the values of each primitive V-function (for each possible set of arguments), the values of each parameter, and the set of existing designators for each designator type. As a result, a mapping function

maps the upper-level primitive V-functions, parameters, and designators down to those of the lower level.

We now discuss how these representation decisions are expressed. For each upper-level primitive V-function

```
VFUN pv(t1 f1;t2 f2; ...) -> tr result;
```

there is a <u>mapping expression</u>

```
pv(t1 f1;t2 f2; ...) : expr
```

The expression on the right in general depends upon entities from the lower-level machine (primitive V-functions, parameters, designator types, and scalar types) and upon pv's formal arguments (f1,f2,...). The type of "expr" is the same as the type of pv's result (here, tr), except as noted below. The intention is for this expression to provide a value for pv (for each possible set of arguments) solely in terms of lower-level entities.

Since "expr" is supposed to deal with lower-level concepts only, how do we interpret occurrences in "expr" of an argument fi if the associated type ti depends on designator or scalar types of the upper-level? The answer is that we must also provide mapping expressions for the upper-level scalar and designator types, mapping them to type specifications that involve only predefined SPECIAL types and types defined by the lower-level modules. An appearance of fi is thus understood to have the <u>mapped</u> type of ti when it appears in "expr".

A similar situation exists with regards to the type of pv's result. If the type of the result depends only on predefined types, then this type and the type of "expr" must be the same; otherwise, if it depends on upper-level types, then the type of "expr" must be the result's <u>mapped</u> type.

The mapping expressions for parameters are just the same as those for primitive V-functions, except that the right side may not reference the lower-level's primitive V-functions.

A. Paragraphs of a Mapping Function Specification

We now present the SPECIAL syntax for mapping function

specification. Like a module specification, a mapping function consists of a number of paragraphs. Some of the paragraphs contain information that is redundant with that in the module specification, but is included in the representation to permit syntactic (including type-consistency) checking. Other paragraphs contain subsidary information that aids in the structuring of the representation specifications. The two major paragraphs are the MAPPINGS and the INVARIANTS paragraphs.

The general form for a mapping function specification is

```
MAP <symbollist> TO <symbollist>;

    TYPES
        <types body>

    PARAMETERS
        <parameters body>

    DEFINITIONS
        <definitions body>

    EXTERNALREFS
        <externalrefs body>

    INVARIANTS
        <invariants body>

    MAPPINGS
        <mappings body>

END_MAP
```

Of course, in a given specification, some of these paragraphs may be missing.

The first <symbollist> gives the names of the modules in the upper-level submachine, the second the names of the lower-level modules.

The TYPES paragraph here has the same syntax and somewhat the same purpose as the TYPES paragraph in a module specification. Though they both give named type definitions, these types are for convenience only; "exportable" types must be defined in a <u>module</u> specification. Thus, new designator or scalar types may not be introduced in a mapping function.

The PARAMETERS paragraph here also has the same syntax as its module specification counterpart. The parameters of a mapping function,

however, serve a somewhat different purpose. Each mapping function parameter denotes a lower-level object (element or function) that is assumed to exist and remain fixed in value throughout the use of the lower-level modules by those at the upper-level. These objects can be freely referred to in other paragraphs of the mapping function. Typically, a parameter is a particular lower-level designator used in representing some upper-level object, or is some function of lower-level designators.

For example, suppose that a lower-level module provides fixed-length character arrays (chararrays); and in the mapping function, we want to assume the existence of a particular chararray, say one that contains character representations of the digits 0-9. Then, the PARAMETERS paragraph would contain the declaration

chararray digits

which declares a particular chararray named "digits". Thereafter in the mapping function we may assume the existence of "digits". If we want a parameter to possess certain properties, we place the appropriate constraints in the INVARIANTS section. In this case, we probably want "digits" to have the properties (1) its indices range from 0 to 9, and (2) the ith position contains the character representation of i.

Of course, some implementation program is required to set-up such parameters and bind them to values consistent with the constraints in the INVARIANTS paragraph.

The DEFINITIONS paragraph is identical in syntax and purpose to its module specification counterpart. Functions are defined here for use within other mapping function paragraphs. The only names that may appear freely in the function body are those of its arguments, lower-level entities, and the function's name itself (for recursive definitions).

The EXTERNALREFS paragraph declares the relevant entities from the upper- and lower-level modules, and has the same syntax as the EXTERNALREFS paragraph in a module specification. All of the upper-level primitive V-functions, parameters, designator types and scalar types must be mentioned here, while only the participating

lower-level ones appear. In addition, we also declare here entities from modules not in the upper or lower representation clusters but which nonetheless are referenced in the mapping specification.

We illustrate a typical situation with an example. In the upper cluster is a module that provides stacks of char_string designators, and in the lower cluster is a module that provides char_string arrays. The way in which char_string stacks are represented in terms of char_string arrays is independent of how char_string designators themselves are represented. Thus, the char_string designator is externally referenced in the representation specification, though the char_string module is not in either the upper or the lower representation cluster. It is, however, in both the upper- and lower-level machines.

The INVARIANTS paragraph is the mapping function counterpart of the ASSERTIONS paragraph for module specifications. Each invariant in the paragraph is a constraint on the values of the lower-level's V-functions, module parameters, and mapping function parameters. These invariants reflect the non-arbitrary way in which the lower-level modules are used. As such, they are typically stronger than assertions which can be proved from the lower-level module specifications alone. Upper-level entities are not permitted to appear in INVARIANT expressions.

It must be proved that the implementation indeed holds each of these constraints invariant. That is, for each implementation of an upper-level operation, we assume that the invariants always hold on entry and then show that they are always TRUE on exit. Finally, we prove that the initialization programs establish the invariants initially. By induction then, these invariants are always true (hence, they indeed are invariants).

Typically, there are many invariants that could be disclosed in this paragraph. It is recommended, however, that the specifier use discretion and record only those invariants that are necessary for proof or that form the basis for simplifications in the abstract programs.

The syntax for the INVARIANTS paragraph is:

<invariants paragraph>: INVARIANTS [<expression>';']+

We require that each <expression> be Boolean-valued.

An example of invariants that characterize the desired properties of the "digits" chararray discussed above are:

```
INVARIANTS
   lbound(digits) = 0  AND  hbound(digits) = 9;
   FORALL INTEGER i INSET {0 .. 9}:
       elt(digits,i) = numchar(i);
```

where numchar is given by the definition

```
CHAR numchar({0..9} i) IS
   IF      i=0 THEN '0'
   ELSE IF i=1 THEN '1'
   ELSE IF i=2 THEN '2'
   ELSE IF i=3 THEN '3'
   ELSE IF i=4 THEN '4'
   ELSE IF i=5 THEN '5'
   ELSE IF i=6 THEN '6'
   ELSE IF i=7 THEN '7'
   ELSE IF i=8 THEN '8'
   ELSE             '9';
```

Finally, there is the MAPPINGS paragraph. It contains a list of mapping expressions, one for each upper-level entity from the EXTERNALREFS paragraph: primitive V-functions, parameters, designator types, and scalar types. The purpose and general form for these mapping expressions is given at the beginning of this chapter.

The syntax equations for the paragraph are:

<mappings paragraph>: MAPPINGS [<map> ';']+

<map>: <vfunmap> | <parmmap> | <typemap>

<vfunmap>: <symbol> <formalargs> ':' <expr>

<parmmap>: <symbol> <formalargs>? ':' <expr>

<typemap>: <symbol> ':' <typespec>

Please refer to the previous MAPPINGS discussion for the appropriate context-sensitive constraints on these equations. The <typespec> in a <typemap> must depend solely on lower-level entities.

B. A Small Mapping Function Example

We now consider how a bounded stack might be represented in terms

of an array. The entities of the (upper) bounded stack module are:

- the designator type "stack"

- the primitive V-functions

```
Access(stack s; INTEGER i) -> INTEGER x;
Size(stack s) -> INTEGER j;
Maxsize(stack s) -> INTEGER i;
```

The relevant array module entities are:

- the designator type "array"

- the primitive V-functions

```
elt(array a; INTEGER i) -> INTEGER x;
lbound(array a) -> INTEGER l;
hbound(array a) -> INTEGER n;
```

In this representation, we intend to associate with each stack an array having lbound no larger than 0 and hbound = Maxsize. The 0'th position in the array holds the stack pointer, which also serves as the representation for Size. Since lbound does not enter into the mappings, its externalref is omitted.

```
    MAP stack_module TO array_module;

    EXTERNALREFS
    FROM stack_module:
stack: DESIGNATOR;
VFUN Access(stack s;INTEGER i) -> INTEGER x;
VFUN Size(stack s) -> INTEGER j;
VFUN Maxsize(stack s) -> INTEGER i;

    FROM array_module:
array: DESIGNATOR;
VFUN elt(array a;INTEGER i) -> INTEGER x;
VFUN hbound(array a) -> INTEGER n;

    MAPPINGS
stack: array;
Access(stack s; INTEGER i):
    IF i INSET {1 .. elt(s,0)} THEN elt(s,i) ELSE ?;
Size(stack s): elt(s,0);
Maxsize(stack s): hbound(s);
END_MAP
```

C. Mapped Specifications

The mapping functions are also used to derive <u>mapped specifications</u> for the upper-level modules. From the mapped specifications it is then possible to derive entry and exit assertions for the programs that implement the upper-level operations. Each such program is next proved <u>correct</u> with respect to its entry and exit assertions. A full discussion of this method of hierarchical proof is contained in [6].

Mapped specifications for a given upper-level module are obtained in the following manner. For V-functions and parameters, mapping expressions follow the schema:

$g(t1\ f1;t2\ f2;...)\ :\ exp_g(f1,f2,...)$

For types, they follow the schema:

$t\ :\ typespec_t$

The mapped specifications are thus formed from a given module specification (relative to a given mapping function) by substituting

1. $exp_g(a1,a2,...)$ for $g(a1,a2,...)$
2. $typespec_t$ for t

The interpretation of $'exp_g(a1,a2,...)$ is that it denotes an expression just like exp except that all V-functions appear quoted. For example,

```
'(IF a=0 THEN vfun(a)+1 ELSE vfun(a)-1)
```

denotes

```
(IF a=0 THEN 'vfun(a)+1 ELSE 'vfun(a)-1)
```

Furthermore, if the result variable of a V-function occurs in the INITIALLY constraint, we embed the constraint in a FORALL expression as follows. If exp is the INITIALLY constraint for vfun(t1 f1;t2 f2;...) and r is the result variable (of type tr), then we replace exp with:

```
FORALL t1 f1;t2 f2;... :
    LET tr r = vfun(f1,f2,...) IN  exp
```

All type definitions from the mapping function specification are also carried over to the mapped specification.

An example of mapped specifications for our familiar bounded stack is given in Appendix A.

VII The Hierarchy Specification Language

HSL (Hierarchy Specification Language) is a simple language for describing how modules are collected together into machines, and how machines are structured into a hierarchy. The former is given by a sequence of INTERFACE clauses, one for each machine; the latter is given by the HIERARCHY clause.

In general, each INTERFACE clause provides the following information about an abstract machine:

1. The machine's name
2. The modules that comprise the machine
3. Any functions, parameters, scalar types, or designator types that are not to be made available to the next higher-level machine. By "available," we mean that such objects may not be called by a program implementing an upper-level module. Note that hidden V-functions are by definition unavailable, so are not included in this list of unavailable objects.

It should be emphasized that a given module may appear in more than one machine, and may make different objects available in different machines.

The syntax for an INTERFACE clause is:

```
<interface>: '(' INTERFACE <symbol> <module info>+ ')'

<module info>: '(' <symbol> [WITHOUT <symbol>+]? ')'
```

The <symbol> in <interface> names the machine; the first one in <module info> gives a module's name; and the ones in the optional WITHOUT part give the names of unavailable objects from the given module.

The HIERARCHY clause contains a list of subclauses, each of which discloses the names of a lower-upper machine pair, and the names of the mapping functions that connect the two machines. These subclauses are ordered from lowest (primitive) to highest level.

The syntax for the HIERARCHY clause is:

```
<hierarchy>: '(' HIERARCHY <symbol> <level>+ ')'

<level>: '(' <symbol> IMPLEMENTS <symbol> USING
             <symbol>+ ')'
```

The <symbol> in <hierarchy> names the specified hierarchy; the first one in <level> gives the name of the lower machine in the lower-upper pair; the second one gives the name of the upper machine in the pair; and the ones in the USING part give the names of the appropriate mapping functions. Currently, the name of a mapping function is given by the name of the file in which it is stored.

The INTERFACE and HIERARCHY clauses are put together into an HSL specification as follows:

<hsl>: <interface>+ <hierarchy>

As an example, consider a two-level hierarchy where the primitive level provides arrays and the second level provides stacks. The HSL specification might be:

```
(INTERFACE stack_machine (stack_module))

(INTERFACE array_machine (array_module))

(HIERARCHY stacker
    (array_machine IMPLEMENTS stack_machine USING stack_array))
```

Note that stack_array.MAP is the name of the file that contains the stack-to-array mapping function.

VIII ILPL Specifications

In this chapter we are concerned with the formulation of implementation decisions. We first discuss our view of the important qualities of an implementation language, and then present our particular implementation language, ILPL.

A. Qualities of an Implementation Language for HDM

Though ILPL has profited greatly from the advances in programming language design, we have also been motivated to eliminate many of the complicated mechanisms and so-called "features" that characterize most current and proposed languages. Some of the complexity has been transferred to other languages of HDM, and some has been omitted altogether.

ILPL is not intended as a stand-alone programming language (though it could be), and at present, no ILPL compiler exists. Rather, ILPL has been designed to express implementation decisions in a framework that supports the concepts and mechanisms of HDM. The actual "running code" for the system is written in a conventional programming language. ILPL is viewed as a language that fits between SPECIAL and such a conventional programming language -- hence the modifier "intermediate".

Some of the following implementation language desiderata relate to programming languages in general, others to an implementation language for HDM in particular.

The language should be simple. Many programming languages suffer from an overabundance of irregular, overly complicated, and poorly understood mechanisms. Our aim has been to eliminate as much as possible from the language while maintaining expressive power and flexibility. Among the troublesome features we have eliminated are pointers, unrestricted side-effects, complicated argument passing disciplines, and built-in synchronization primitives. Whenever a need for one of these mechanisms has occurred, we have been able to provide an appropriate module that supports the desired mechanism.

The language should have a mathematical semantics. Modern programming languages should be designed with formal proof in mind. One approach is axiomatization, that is, to provide a proof rule for each construct in the langua . Those features that are difficult to understand and use are often, not surprisingly, the most difficult to axiomatize. Such features are best left out of the language. Although we are only now writing proof rules for ILPL, we have attempted to include only those constructs that are easily axiomatized.

The language should support the writing of "structured programs". It is now well-accepted that structured programs are easier to write, read, modify, and prove. HDM, if used properly, should lead to modular, provably correct programs. HDM also supports the more narrow "goto-less" view of structured programming. Besides the usual if-then-else and loop control constructs, ILPL also provides a structured exception handling mechanism, and omits the "goto" statement.

The language should be fully integrated into HDM. Although any language can, in principle, be used for implementation in HDM, an intermediate-level HDM-based implementation langauge can narrow the (often frighteningly wide) gap between specification and running code. As integrated into HDM, ILPL programs incorporate HDM concepts and mechanisms both at the specification and at the expression level. ILPL programs can also be checked for consistency with specifications from other stages, in particular, with hierarchy descriptions, module specifications, and representation specifications.

The language should be translatable to efficient code. The abstract ILPL programs are intended to be specifications for the ultimate implementation code. It should be possible, nevertheless, to translate automatically an ILPL specification with relative ease into either a common high-level language or an assembly language. As a result, we have left out of ILPL those mechanisms that cannot be implemented efficiently. In addition, there are some optimizations that are particularly appropriate to ILPL programs, such as elimination of impossible exception conditions and in-line expansion of calls to small subroutines.

B. Overview of ILPL

In HDM, each module in the hierarchy is implemented by a collection of ILPL programs. When the user invokes a visible operation of the top-level machine, he sets into motion a chain of cascading invocations: the top-level invocation precipitates a sequence of invocations of visible functions or parameters of (1) modules at the next lower level, or (2) modules at the same level but "below" the given one according to the partial ordering induced by the external reference relation. Each such invoked operation itself precipitates a sequence of invoked operations, and so on until the operations of the primitive machine are invoked and executed.

Each module implementation includes:

- programs for each of the module's visible functions
- an INITIALIZATION program
- programs that serve as private subroutines which cannot be invoked by programs outside of the given collection.

The INITIALIZATION program is to be executed before any operations of the implemented module are invoked. Its purpose is to drive the module to its initial state. In doing so, it may invoke operations of the <u>lower</u> implementing modules, but not of the <u>upper</u> implementing ones. Thus, assuming that the primitive machine is in its initial state when "powered up," we see that the INITIALIZATION programs are executed in bottom-up order, starting with level 2.

More precisely, a module is in its initial state if the values obtained by applying the representation functions for the module's state-functions (primitive V-functions and parameters) to the values of the next-lower level's state-functions satisfy the initial value constraints from the module's mapped specification.

For a module that appears at multiple adjacent levels, only the lowest-level appearance need actually be implemented.

C. The Expression Level of ILPL

With the below exceptions, the expression level (types, type operations, and expression forms) of ILPL is exactly the same as that of SPECIAL. In this way, expression level concepts of SPECIAL carry over directly to ILPL.

The difference is that in SPECIAL an expression is a mathematical denotation of a value; there is no notion of "computing the value" of a mathematical denotation. On the other hand, the same expression in ILPL has an underlying implementation that when <u>executed</u> computes the denoted value; the notion of computation here is inherent. Such considerations have caused us to omit from ILPL those types, operations, and expression forms of SPECIAL that in their full generality cannot be implemented efficiently. Specifically, we have omitted (1) the set data type and its associated operations, and (2) quantified expressions (i.e., FORALL and EXISTS). Set types involve objects of possibly infinite cardinality, and quantified expressions involve quantifications over possibly infinite domains. Unfortunately, practical <u>implementations</u> for these very useful <u>specification</u> objects do not exist. If the designer wants finite versions of these objects, he can easily specify them as modules and provide his own (practical) implementations.

Since ILPL is an imperative language, with sequencing, local variables and assignment statements, the characterization expressions (LET and SOME) have been rendered unnecessary, and thus have also been omitted from ILPL.

All the other types, type operations, and expression forms from SPECIAL have remained intact in ILPL, with all the same type-checking rules. The following table lists the ILPL types and expression forms.

primitive types	INTEGER, REAL, BOOLEAN, CHAR, subtypes (including subranges), scalar types, and designator types
constructed types	vector, structured, and union types
expression forms	IF-THEN-ELSE

The operations on the above types are the same as those mentioned in Section IV.D.

Although set types have been eliminated, there is still a need for set expressions (i.e., set constructors) in the type specifications for subtypes and scalar types. This is, however, the only place in an ILPL specification where set expressions may appear.

D. The Specification Level of ILPL

Like SPECIAL specifications, an ILPL specification consists of a number of paragraphs. In its most general form, the top-level structure looks like:

```
IMPLEMENTATION <symbol> IN_TERMS_OF <symbollist>;

    TYPES
        <types body>

    PARAMETERS
        <parameters body>

    EXTERNALREFS
        <externalrefs body>

    TYPEMAPPINGS
        <typemappings body>

    INITIALIZATION
        <initialization body>

    IMPLEMENTATIONS
       <implementation body>

END_IMPLEMENTATION
```

The <symbollist> in the header's IN_TERMS_OF clause gives the names of the implementing modules from the lower-level only. The EXTERNALREFS paragraph lists the names of all implementing modules, upper and lower.

1. The "Informational" ILPL Paragraphs

In the TYPES paragraph, the designer declares the "internal" named types that are used in the implementation. The types declared here are similar to those declared in the TYPES paragraph of a mapping function:

they serve mainly to enhance the readibility of the implementation and cannot introduce new designator types. The designator and scalar types from the implemented and implementing modules are declared in the EXTERNALREFS paragraph.

The PARAMETERS paragraph here is a copy of the PARAMETERS paragraph from the mapping function for the representation cluster containing the given module as an upper module. These parameters are constant values and functions of the lower level that are used to represent the upper-level module and must now be implemented. The implementations for these parametric values and functions appear respectively in the INITIALIZATIONS and IMPLEMENTATIONS sections.

Implementation parameters often name designators from the lower level. Though the value of such a parameter may not change (i.e., the parameter may not be on the left-side of an assignment), the object represented by that designator, however, may in fact be modified. That is, the parameter may appear as an actual argument to an O- or OV-function that modifies the object's state. Thus, the parameter itself remains constant, though the state-functions that depend on the designator's type may change as indirect effects are achieved.

The EXTERNALREFS paragraph here is similar in both form and meaning to the previous EXTERNALREFS paragraphs. From the implemented module are declared here all visible operations, designator types and scalar types. From the implementing modules are declared only those entities actually used in the implementation. All modules that appear in the EXTERNALREFS paragraph but not in the implementation's header are assumed to be modules from the same level as the implemented module. The objects typically externally referenced from such modules are designator types and parameters. The information regarding which implementing modules are "upper" and which are "lower" is used by the verification system.

The TYPEMAPPINGS paragraph includes the mapping expressions for the implemented module's designator and scalar types, as previously given in the mapping function for the representation cluster containing the given module as an upper module. In addition, it contains type mappings for

externally referenced types for which knowledge of their representations is required by the implementation. The type mappings are applied in the implementation programs in the same way they are used to obtain mapped specifications. For example, if "stack: array" is a type mapping in a given implementation, then all occurrences in program bodies of variables of type stack are understood to actually have type array.

As an example of a case where knowledge of an externally referenced type's representation is required, consider the implementation of a module that provides stacks of char_strings in terms of HDM vectors. Let's also assume that the char_string module appears at the stack module's level. Now, if the char_string module does not also appear at the lower level, then the stack implementation must know how char_string designators are represented. On the other hand, if the char_string module <u>is</u> at the lower level, then the stack implementation need <u>not</u> have this information. The intended representation in this case is "VECTOR_OF char_string". In both cases, the char_string designator is externally referenced in the stack implementation, though only in the former case does a type mapping for the char_string designator appear.

The syntax of the TYPEMAPPINGS paragraph is thus a restricted form of the syntax for a MAPPINGS paragraph:

```
<typemappings paragraph>: TYPEMAPPINGS [<typemap> ';']+
```

2. ILPL Programs

In the remainder of this chapter we discuss the last two paragraphs in an ILPL specification. These paragraphs give the implementation's programs.

The general structure of an ILPL program is:

```
<pgm header>
DECLARATIONS
    <declaration list> ';'
BEGIN
    <stmtgroup>
END;
```

The header identifies the program's purpose, gives the program's name, and declares its formal arguments and result.

The simplest header is

INITIALIZATIONS

which identifies the program as the one that initializes the module's state. Initialization involves: (1) invoking programs of the lower level to initialize the module's V-functions, and (2) assigning values to value parameters - both parameters from the implemented module and parameters from the implementation's PARAMETERS paragraph (i.e., representation parameters). Value parameters retain their values between invocations; that is, they are not like local variables.

For each (visible) module operation there is a program that implements that operation. For each O-function there corresponds an O-program with header

OPROG <symbol> <formalargs> ';'

where <symbol> is the name of the implemented O-function, and <formalargs> matches the given function's formal argument list in its module specification. Similarly, for each OV-function there is an OV-program with header

OVPROG <symbol> <formalargs> '->' <resultarg> ';'

and for each (visible) V-function and parametric function there is a V-program with header

VPROG <symbol> <formalargs> '->' <resultarg> ';'

VPROGS are also used to implement parametric functions. Again, the formal and result argument declaration lists in the above headers should match the ones in the function's module specification.

The other headers are for subroutines that are callable by other programs in the implementation but not by programs outside the given implementation. A subroutine that may cause a state change (i.e., one that may call O- and OV-programs of other modules) but does return a value is called an OSUBR. Similarly, a subroutine that invokes only V-programs of other modules and returns a value is called a VSUBR. Finally, a subroutine that may both cause a state change and return a value is an OVSUBR.

The DECLARATIONS section in an ILPL program declares the program's

local variables. It is understood that each time the program is invoked these variables have no value, i.e., they do not retain values between invocations. Recall, however, that V-functions do retain values between invocations. There is no inner scope in a program beyond the DECLARATIONS section in which variables can be declared.

Between the BEGIN and END brackets is a list of program statements, called a <stmtgroup>. In general, these statements may reference PROGs and parameters other modules and SUBRs and implementation parameters of the given module.

```
<stmtgroup>: [<stmt> ';']+
```

We now discuss the types of ILPL statements. The most trivial statement is the empty statement. The most basic "real" statement is the assignment statement, of the general form:

```
<lhs> '<-' <expression>
```

The expression on the right-hand side may contain applications of value-returning programs. The type of the right-hand side must be compatible with the type of the left. The left-hand side of an assignment may be a variable, a structure component, or a vector element. Syntactically,

```
<lhs>: <symbol>
     | <lhs> '.' <symbol>
     | <lhs> '[' <expr> ']'
```

Of course, the <expr> in the last alternative must be integer-valued.

A program invocation may also appear as a basic statement by itself, like a procedure call in Algol 60 or Pascal. If the invoked program is value-returning, its returned value is discarded.

Programs that implement operations, however, may in general return from an invocation with an exception. We have provided a mechanism whereby such programs can be invoked and exceptions be handled in a structured manner. This mechanism is the EXECUTE statement. For example, if f is the name of an OV-function with exceptions ex1, ex2, and RESOURCE_ERROR, the program implementing f could be called as follows:

```
EXECUTE x <- f(y) THEN
    ON ex1: ...
    ON ex2: ... ;
    ON RESOURCE_ERROR: ... ;
    ON NORMAL: ... ;
END;
```

On a normal return, x is assigned the returned value of f(y) and the statements in the NORMAL handler are executed; otherwise, for an exception return, the statements in the corresponding exception handler are executed.

A similar format is used for invocations of other function types. If the invoked function does not return a value (i.e., an 0-function), or the returned value is not to be retained, the left part of the assignment is omitted.

A given exception handler may be omitted if the corresponding exception can never occur. If such a handler is omitted and the corresponding exception <u>does</u> occur, then the program aborts. The NORMAL handler may be omitted anytime no special action is desired on normal return. If omitted and a normal return occurs, execution continues at the following statement (i.e., the one after the EXECUTE statement).

The EXECUTE format should be used whenever it is possible for an invoked operation to have an exception return. If no exception returns are possible, the operation can be invoked in the more conventional manner.

The syntax for the EXECUTE statement is:

```
<execute stmt>: EXECUTE
                    [<lhs> '<-']? <functioncall> THEN
                <execute handler>+
                END

<execute handler>: ON <xeq event list> ':' <stmtgroup>

<xeq event list>: <xeq event> [',' <xeq event>]*

<xeq event>: <symbol> | RESOURCE_ERROR | NORMAL
```

A given event may occur in only one handler. Furthermore, execute events besides NORMAL must be valid exception names from the invoked function's specification.

Now that we know how to invoke a program, we next discuss how to return from one. A program causes an exception return with the statement

```
RAISE '(' [<symbol> | RESOURCE_ERROR] ')'
```

Again, the name of a raised exception must be a valid exception name for that operation. A normal return is effected either by "falling off the end" of the program, or by executing

```
RETURN;
```

For value-returning programs, the value returned is the final value of the formal result argument.

In recent years there have been extensive pleas for "structured" control constructs in programming langugages. The primary target has been the "goto" statement, which potentially allows for arbitrary jumps. Undisciplined use of the goto often leads to "rat's nest" programs that are extrememly difficult to understand. Instead, it has been suggested that programs be realized as a proper nesting of statement constructs.

Besides the usual if-then-else and loop "structured" control constructs, ILPL also provides event-driven statements along the lines proposed by Zahn [8]. The first such construct allows for the sequential execution of a group of statements that may be "quick exited" by "signalling" one of a set of "events". This construct has the form:

```
EXPECTING e1,e2 DO
    ...
    SIGNAL(e1);
    ...
    SIGNAL(e2);
    ...
THEN
    ON e1: ... ;
    ON e2: ... ;
    ON NORMAL: ... ;
END;
```

The first line declares the list of events that are applicable for the given statement. Here, there are two events, e1 and e2. For each declared event there is a corresponding event handler after the construct's statement body (i.e., after the THEN). If a SIGNAL statement in the statement body is executed, control is transferred

immediately to the corresponding event handler; on the other hand, if the statement body is executed without hitting a SIGNAL, control goes to the NORMAL handler. If the corresponding handler for a given event is missing when that event is signalled, or, for the NORMAL handler, when control falls through the body, then control continues at the statement following the END. If no handlers at all are provided, then the THEN is omitted, too.

The syntax for the EXPECTING statement is thus:

```
<expecting stmt>: EXPECTING <symbollist> DO
                      <stmtgroup>
                  [THEN <event handler>+]?
                  END

<event handler>: ON <eventlist> ':' <stmtgroup>

<eventlist>: <event> [',' <event>]*

<event>: <symbol> | NORMAL
```

The next event-driven statement is an event-driven repeat-until loop. The UNTIL statement is similar to the EXPECTING statement, except that its statement body is repeatedly executed until one of its declared events is signalled. Since an explicit signal is required to exit the loop, it is meaningless (hence, not permitted) to have a NORMAL handler for an UNTIL statement. The syntax for the UNTIL statement is the same as for the EXPECTING statement, except (1) UNTIL naturally replaces EXPECTING, and (2) as mentioned, NORMAL is not allowed as an event name in a handler.

Event statements may be nested, and each nesting introduces a new scope of event names. A SIGNAL statement within an event statement body may only signal an event from the innermost active scope. Since the event handlers for a given event statement are outside of its statement body, they may signal events from the next outer scope. For example, consider the following fragment:

```
UNTIL e1,e2 DO
    EXPECTING e1,e3 DO
        ...
        SIGNAL(e3);
        ...
        SIGNAL(e1);
        ...
    THEN
        ON e1: ... ;
        ON e3: SIGNAL(e2);
        ON NORMAL: ... ;
    END;
THEN
    ON e1: ... ;
    ON e2: ... ;
END;
```

The signalling of e3 in the inner scope causes e2 in the outer scope to be signalled. Note that e1 in the inner scope is a different event altogether from the e1 in the outer scope.

ILPL also provides a standard FOR loop, which looks like:

```
FOR i FROM expr1 BY expr2 TO expr3 DO
    <stmtgroup>
END
```

The counting variable (here, i) and all of the range expressions (here, expr1, expr2, and expr3) must have INTEGER type. The BY part may be omitted, in which case the counting variable is incremented by 1 each iteration. The semantics of the FOR loop may be described by the following code segment:

```
i <- expr1;
i2 <- expr2;
i3 <- expr3;
UNTIL loopexit DO
    IF i > i3 THEN SIGNAL(loopexit); END_IF;
    <stmtgroup>
    i <- i + i2;
END;
```

The FOR and the UNTIL statements can be combined to make a FOR-UNTIL statement, which is an UNTIL statement with a FOR header grafted onto the front. In the FOR-UNTIL statement, the TO part of the FOR header is optional; if omitted, the loop is executed until some event is explicitly signalled. As long as the TO part appears, a FOR-UNTIL statement is allowed to have a NORMAL handler, which applies

to loop exits caused by the index variable falling out of range.

The syntax of the FOR-UNTIL statement is:

```
<for_until stmt>: <for_untilhead> DO
                      <stmtgroup>
                  [THEN <event handler>+]?
                  END

<for_untilhead>: <forhead> UNTIL <symbollist>

<forhead>: FOR <symbol> FROM <expr> [BY <expr>]?
                        [TO <expr>]?
```

As mentioned, ILPL also provides an if-then-else statement. It contains a single THEN clause, followed by an arbitrary number of ELSIF clauses, followed by an optional ELSE clause. We saw a simple example in the above code fragment for the FOR loop semantics. The syntax of the if-then-else in ILPL is:

```
<if stmt>: IF <ifclause> <elseclause> END_IF

<ifclause>: <expr> THEN <stmtgroup>

<elseclause>: [ELSIF <ifclause>]*  [ELSE <stmtgroup>]?
```

Of course, we require the <expr> in <ifclause> to be Boolean-valued.

The last statement type in ILPL is the TYPECASE statement, based on the TYPECASE expression form discussed in Section 4.4.5. It allows for structured transfer of control based on the type of the expression that is given as an operand. The statement has the syntax:

```
<typecase stmt>: TYPECASE <expr> OF <caselist> END

<caselist>: [<typespec> ':' <stmtgroup>]+
```

Complete examples of ILPL specifications are contained in Volume III.

E. Argument Passing in ILPL

The semantics of argument passing in ILPL is quite straightforward, namely <u>call-by-value</u>. Some explanation is needed, however. Consider, for example, a program with the following header:

```
OPROG pushi(stack s; INTEGER i);
```

The formal arguments correspond respectively to the (name of the) selected stack, and an integer to be pushed. If "pushi" is invoked with:

```
pushi(s1,j)
```

it can be viewed that the following assignments are made upon invocation.

```
s <- s1;
i <- j;
```

That is, the arguments to "pushi" are evaluated once, and their values copied onto the two local variables s and i inside the body of "pushi". Any reference to s and i in the body of "pushi" is hence to the copies, and assignments to s or i do not affect s1 or j, as would be the case with call-by-reference. It is permitted in ILPL, though in general not recommended, to assign to formal arguments.

Although the calling discipline is call-by-value, thus precluding modification to s1 itself in "pushi", it is important to note that the primitive V-functions of the array module, which are used to implement stacks, can be modified by calls to array O- and OVPROGs in the "pushi" body. For example, if we represent a stack by an array with the stack pointer in position 0 of the array, the code for the implementation of "pushi" might be:

```
OPROG pushi(stack s; INTEGER i);
BEGIN
    change(s,0,elt(s,0)+1);
    change(s,elt(s,0),i);
END;
```

(Note that we have disregarded exceptions here). As a result of the calls to the O-function "change", the values of the primitive V-function "elt" of the array module have changed. Thus side-effects are allowed, but only indirect ones through the invocation of O- or OV-functions of the lower-level machine.

IX The Tools of HDM

The tools of HDM create a (partial) environment for manipulating system specifications. The tools themselves are written in INTERLISP under TOPS-20 or TENEX. The most relevant aspect of this implementing environment for the tools user is the file handling subsystem. The user should be familiar with how files are created and manipulated on the system (TOPS-20 or TENEX) on which his version of the tools resides.

On these systems a file name has the form:

<DIR>PRE.EXT.n

where

DIR is the directory,
PRE is the prefix,
EXT is the extension,
n is the version number.

A. General Presentation of the System

The components of a system specification, as we have discussed, are:

- An HSL description, which describes how modules are collected together into machines, and how machines are organized hierarchically into a system.
- Module specifications for all modules in the system.
- Mapping functions for all representation clusters.
- ILPL implementations for all modules.

Generally, each individual specification must satisfy certain consistency criteria. Internal consistency criteria are concerned with self-contained properties of the specification, while external consistency criteria relate to properties of the specification considered in a larger context.

Our system consists of various functional units which perform the necessary checks. The system is set-up as an INTERLISP subsystem containing commands for each functional unit in the environment. The

environment is invoked at the operating system command level by typing the subsystem's name. For example, on the SRI-KL machine where the subsystem is stored in <HIER>NS.EXE, the subsystem is invoked by typing

<HIER>NS

The functional units are the module handler, the mapping function handler, the implementation program handler, the interface handler, and the hierarchy handler. Each takes as input one or more files and produces either error diagnostics or a "link" file, which records the successful completion of the particular set of consistency checks.

Typically, a link file contains information about the processed file that needs to be consulted when other specifications, say for other modules in the same machine, are processed. For example, the link file would typically include a list of a module's external references. As a rule, a link file is produced only when all the relevant checks have been carried out and met: the link file's existence is prima facie evidence of the correctness of some part of the system. However, it may be the case that after having checked, say, an interface, the user changes the specification for some contained module without re-checking the interface. The source file for the new module will thus have been created later than the link file for the interface. This is a typical case of inconsistent files, and will be detected automatically. It can be remedied by performing all the necessary interface checks to create up-to-date link files.

B. The Module Handler

The module handler expects a source file containing a SPECIAL module specification. The command that checks a module specification's internal consistency is

CHECKMODULE(modulename)

where

modulename.SOURCE

is the file containing the desired module specification. In verifying that the given specification complies with the rules of SPECIAL, the

command operates in two phases, the first to check for context-free errors, the second to check for context-sensitive errors. Sometimes these phases are called the "syntactic" and the "semantic" phases, respectively.

The first phase passes the specification through a context-free parser for SPECIAL module specifications. If a syntax error is encountered, an error message is printed that displays the invalid syntactic token between !!!...!!! in a "window" of 100 characters of the source text. At this point the user (currently) has three choices for error recovery:

1. R - Replace the offending lexeme by an arbitrary string. This will cause a new version of the file to be generated and the parse to be restarted at the point of error.

2. T - Teco. This will cause the TECO text editor to be invoked as a subprocess, the source file to be yanked, and the "cursor" to be positioned before the first character of the offending lexeme. The user can then arbitrarily modify the file. When the editor is exited, control is returned to CHECKMODULE where the parsing may continue: (a) at the point of error - option D, for default, (b) at the beginning of the file - option S, for start over, or (c) at an arbitrary location in the file, specified by a relative byte address in the file. This latter approach is very unreliable because the parser does not backtrack its internal state to that point.

3. A - Abort. This will abort the parse and go into error mode.

If at any time the user is unsure of how a prompt may be answered, he should type "?", which will cause the various permissible responses to be listed and explained.

When the entire file has been successfully parsed, the full file name is printed and the second phase is initiated.

The second phase is mostly concerned with the enforcement of the type rules, scope rules, and other context-sensitive constraints. Various error or warning messages may be issued, giving some information about the location of the error, the kind of error, and sometimes context information in a prefix form of the original source expression. For example,

```
a + b
```

would appear as

(+ a b)

If no errors were detected during the second phase, the user is asked whether he wants a link file made. If the answer is Y, a link file modulename.MLINK will be written, and its name returned as the value of CHECKMODULE; if the answer is N, no link file will be created and CHECKMODULE returns with value T.

The link file is used by the interface handler to check the module's external references against their original definitions.

The other command in the module handler is

REFORMAT(inputfilename outputfilename)

This causes the module specification contained in inputfilename to be pretty-printed on the file outputfilename. If outputfilename is omitted, a new version of the input filename is created and serves as the output filename. For example,

REFORMAT(STACK.SOURCE)

will cause the specification in STACK.SOURCE to be pretty-printed on a file whose name is a new version of STACK.SOURCE.

Note that any file can be typed at the terminal from within the tools by the command "TTY filename".

C. The Mapping Function Handler

The mapping function handler is separated into two functional units, one that checks internal consistency, the other external consistency.

The internal consistency checker CHECKMAPSPEC is the counterpart of CHECKMODULE for mapping functions. It is invoked with

CHECKMAPSPEC(mappingfnname)

where the desired source file has filename mappingfnname.MAP. Similar to CHECKMODULE, CHECKMAPSPEC consists of two phases: a context-free parse, and a context-sensitive check. The link file here is given filename

mappingfnname.SYNLINK

Mapping function specifications may also be pretty-printed with the REFORMAT command.

The external consistency checker is invoked with the command

CHECKMAP(name)

and will attempt to take its input from the link file name.SYNLINK. The checks are:

- File system consistency: each of the modules referenced in the corresponding mapping function should have been previously checked -- which means there should be MLINK files for them -- and the corresponding source files should not have been modified since then.

- External type consistency: all the objects that have been declared external in the mapping function specification are checked against their original declaration in the module specifications.

- Completeness: there must exist a unique mapping expression for each primitive object of the "upper" module. This check can fail in three ways: an object may have no representation; there may be an object appearing in the mapping function specification which is not a primitive object of the "upper" module; or a primitive object may have more than one representation.

If all the checks are performed correctly, a link file is created under the filename name.SEMLINK.

D. The Implementations Checker

Similar to the mapping function checker, the implementations checker contains functions to check both the internal and external consistency of ILPL programs. At the time of this writing, the implementations checker is only partially implemented. As a result, some of the details described in this section may change. Users of the tools will be notified, however, of any such changes.

The first command,

CHECKIMPLSPEC(implname)

takes its input from file implname.IMPL and checks for context-free errors in its first phase and for context-sensitive errors in the second

phase. The name of the generated link file here is implname.ISYNLINK.

The second command,

CHECKIMPL(name)

takes its input from name.ISYNLINK and performs analogous checks as CHECKMAP.

E. The Interface Handler

This checker deals with interface specifications. We take the syntax for an interface specification from the HSL description in Chapter 7.

<interface>: '(' INTERFACE <symbol> <module info>+ ')'

<module info>: '(' <symbol> [WITHOUT <symbol>+]? ')'

The <symbol> in <interface> names the machine; the first one in <module info> gives a module's name; and the ones in the optional WITHOUT part name objects from the given module that are not to be made available to the next higher level.

Filenames for interface specifications should have extension INTERFACE.

The command

CHECKINTERFACE(interfacename)

will cause the interface in file interfacename.INTERFACE to be checked for the following criteria:

- The specification must be well-formed syntactically.
- The prefix of the filename must be the same as the interface name.
- All modules appearing in the interface specification must have been previously specified and checked, and should not have been modified after the check.
- No two objects from modules in the given interface may have the same name.
- The objects on a WITHOUT list must have been defined in the specification of the corresponding module.

- The interface must be closed under external references. That is, all externalrefs of a given module in the interface must be to other modules in the interface. In addition, the types of the externalrefs must be compatible with the declared types in the defining module.

If all these checks are carried out correctly, a link file interfacename.ILINK is created.

It is also worth noting that this command follows the "Do as much as you can" philosophy: if a module has not been specified, all the checks that can be made with the available information are still performed. However, the interface specification will still be considered incomplete, and no link file will be produced.

F. The Hierarchy Handler

Within a hierarchy in HDM, two interfaces are connected if there is a set of mapping functions that implements all the modules of one interface in terms of objects that are visible in the other. A hierarchy is thus specified by giving the names of all the interfaces and mapping functions.

```
<hierarchy>: '(' HIERARCHY <symbol> <level>+ ')'

<level>: '(' <symbol> IMPLEMENTS <symbol> USING
             <symbol>+ ')'
```

The <symbol> in <hierarchy> names the specified hierarchy; the first one in <level> gives the name of the lower machine in the lower-upper pair; the second one gives the name of the upper machine in the pair; and the ones in the USING part give the filename prefixes for the files that contain the appropriate connecting mapping functions. The lower machine of the <u>first</u> level clause is the hierarchy's primitive machine. The command

```
CHECKHIERARCHY(hierarchyname)
```

takes its input from file *hierarchyname*.HIERARCHY and performs the following checks.

- The hierarchy specification must be syntactically correct.

- Each lower machine in a level clause must be either the

primitive machine or the upper machine for some previous level clause. Furthermore, each machine except the root must appear once and only once as the upper machine in some level.

- For each level clause, link files must exist for both interfaces and for all mapping functions.

- Each level clause must be consistent and complete.

A given level clause is consistent when the upper modules of the named mapping functions exist in the upper machine, and the lower modules in the lower machine. The clause is complete when all modules in the upper machine serve as upper modules in the level's mapping functions. Note that some mappings may be specified implicitly: these are identity mappings for modules which appear in both the upper and lower machines but which are not named explicitly as an upper module in the level's mapping functions. Such identity mappings need not be actually specified.

Note that a module may appear in several machines, and that a mapping function specification may be used in several level clauses.

Successful completion of these checks will cause the link file hierarchyname.HLINK to be created. Although this file does not link to *any particular treasure*, it contains interesting information about the files that have been used in order to check the hierarchy, and can be helpful for restoring the consistency of the file system.

G. Command Abbreviations

For convenience, all the commands can be invoked with an abbreviated form. As presented, commands are invoked with the form

command(arg)

The abbreviated form uses an abbreviated command name and drops the parentheses. Its form is:

abbrev_command arg

Note that all commands except REFORMAT take a single argument; REFORMAT takes a second optional argument. The system will prompt for a missing argument.

The abbreviations are:

CMO for CHECKMODULE
CMS for CHECKMAPSPEC
CMA for CHECKMAP
CIF for CHECKINTERFACE
CHR for CHECKHIERARCHY
REF for REFORMAT

REFERENCES

[1] Boyer, Robert S. and J Strother Moore.
A Computational Logic.
Academic Press, to appear.

[2] Boyer, Robert S. and J Strother Moore.
A Formal Semantics for the SRI Hierarchical Program Design Methodology.
Technical Report, SRI Computer Science Laboratory, November 1978.

[3] Guttag, John V.
The specification and application to programming of abstract data types.
PhD thesis, Department of Computer Science, University of Toronto, 1975.
Computer Science Research Group Tech. Report CSRG-59.

[4] Hoare, C.A.R. and N. Wirth.
An axiomatic definition of the programming language Pascal.
Acta Informatica 2(4):335-355, 1973.

[5] Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson.
A Provably Secure Operating System: The System, Its Applications, and Proofs.
Technical Report, SRI Computer Science Laboratory, February 1978.
Final report project 4332.

[6] Robinson, Lawrence and Karl N. Levitt.
Proof techniques for hierarchically structured programs.
Communications of the ACM 20(4):271-283, April 1977.

[7] Wensley, J. H., L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock.
SIFT: The design and analysis of a fault-tolerant computer for aircraft control, pages 1240-1255.
IEEE, October, 1978.

[8] Zahn, C. T.
A control statement for natural top-down structured programming.
Proc. Symposium on Programming Languages.

A. Mapped Specifications for a Bounded Stack

```
MODULE bounded_stack_module

    TYPES
stack: DESIGNATOR;

    FUNCTIONS
VFUN Access(stack s; INTEGER i) -> INTEGER elem;
    HIDDEN;
    INITIALLY  elem = ?;

VFUN Size(stack s) -> INTEGER i;
    HIDDEN;
    INITIALLY  i = 0;

VFUN Maxsize(stack s) -> INTEGER j;
    INITIALLY  j = ?;

OVFUN Create_stack(INTEGER bound) -> stack s;
    EXCEPTIONS
        bad_bound:  bound < 1;
        RESOURCE_ERROR;
    EFFECTS
        s = NEW(stack);
        'Maxsize(s) = bound;

OFUN Push(stack s; INTEGER elem);
    EXCEPTIONS
        full:  Size(s) = Maxsize(s);
    EFFECTS
        'Access(s,'Size(s)) = elem;
        'Size(s) = Size(s) + 1;

OVFUN Pop(stack s) -> INTEGER elem;
    EXCEPTIONS
        empty:  Size(s) = 0;
    EFFECTS
        'Size(s) = Size(s) - 1;
        elem = Access(s,Size(s));
        'Access(s,Size(s)) = ?;

VFUN Top(stack s) -> INTEGER elem;
    EXCEPTIONS
        empty:  Size(s) = 0;
    DERIVATION
        Access(s,Size(s));

END_MODULE
```

```
MAP bounded_stack_module TO array_module;

    EXTERNALREFS
    FROM bounded_stack_module:
stack: DESIGNATOR;
VFUN Access(stack s; INTEGER i) -> INTEGER x;
VFUN Size(stack s) -> INTEGER j;
VFUN Maxsize(stack s) -> INTEGER i;

    FROM array_module:
array: DESIGNATOR
VFUN elt(array a; INTEGER i) -> INTEGER x;
VFUN hbound(array a) -> INTEGER h;

    MAPPINGS
stack: array;
Access(stack s; INTEGER i):
    IF i INSET {1 .. elt(s,0)} THEN elt(s,i) ELSE ?;
Size(stack s): elt(s,0);
Maxsize(stack s): hbound(s);

END_MAP
```

The mapped FUNCTIONS specifications for bounded_stack are:

```
VFUN Access(stack s; INTEGER i) -> INTEGER elem;
    HIDDEN;
    INITIALLY
        FORALL stack s; INTEGER i :
            (IF i INSET {1..elt(s,0)}
                 THEN elt(s,i) ELSE ?)
             = ?;

VFUN Size(stack s) -> INTEGER i;
    HIDDEN;
    INITIALLY
        FORALL stack s : elt(s,0) = 0;

VFUN Maxsize(stack s) -> INTEGER j;
    INITIALLY
        FORALL stack s : hbound(s) = ?;

OVFUN Create_stack(INTEGER bound) -> stack s;
    EXCEPTIONS
        bad_bound: bound < 1;
        RESOURCE_ERROR;
    EFFECTS
        s = NEW(array);
        'hbound(s) = bound;

OFUN Push(stack s; INTEGER elem);
    EXCEPTIONS
        full: elt(s,0) = hbound(s);
    EFFECTS
        (IF 'elt(s,0) INSET {1..'elt(s,0)}
            THEN 'elt(s,'elt(s,0)) ELSE ?)
            = elem;
        'elt(s,0) = elt(s,0) + 1;

OVFUN Pop(stack s) -> INTEGER elem;
    EXCEPTIONS
        empty: elt(s,0) = 0;
    EFFECTS
        'elt(s,0) = elt(s,0) - 1;
        elem = (IF elt(s,0) INSET {1..elt(s,0)}
                   THEN elt(s,elt(s,0)) ELSE ?);
        (IF elt(s,0) INSET {1..elt(s,0)}
            THEN elt(s,elt(s,0)) ELSE ?)
            = ?;

VFUN Top(stack s) -> INTEGER elem;
    EXCEPTIONS
        empty: elt(s,0) = 0;
    DERIVATION
        (IF elt(s,0) INSET {1..'elt(s,0)}
```

```
THEN 'elt(s,elt(s,0)) ELSE ?);
```

B. Grammars for SPECIAL and ILPL

```
<special>          : MODULE <symbol> <fparagraphlist>
                   | MAP <symbollist> TO <symbollist> ';'
                       <mparagraphlist>

<actuallist>       : '(' ')'
                   | '(' <expressionlist> ')'

<addop>            : '+'
                   | UNION
                   | DIFF

<aexp>             : <aexp> <addop> <fact>
                   | <aexp> <addop> <symbol>
                   | <symbol> <addop> <fact>
                   | <symbol> <addop> <symbol>
                   | <fact>
                   | <aexp> '-' <fact>
                   | <symbol> '-' <fact>
                   | <aexp> '-' <symbol>
                   | <symbol> '-' <symbol>

<assertions>       : ASSERTIONS <listofexpressionsemi>

<bexp1>            : <bexp1> OR <bexp2>
                   | <symbol> OR <bexp2>
                   | <bexp1> OR <symbol>
                   | <symbol> OR <symbol>
                   | <bexp2>

<bexp2>            : <bexp2> AND <bexp3>
                   | <symbol> AND <bexp3>
                   | <bexp2> AND <symbol>
                   | <symbol> AND <symbol>
                   | <bexp3>

<bexp3>            : <not> <bexp3>
                   | <not> <symbol>
                   | <bexp4>

<bexp4>            : <aexp> <relop> <aexp>
                   | <symbol> <relop> <aexp>
                   | <aexp> <relop> <symbol>
                   | <symbol> <relop> <symbol>
                   | <aexp>

<call>             : ''' <call>
                   | <symbol> <actuallist>
```

```
<case>                    : <typespecification> ':' <expression>

<caselist>                : <case> ';'
                          | <case> ';' <caselist>

<declaration>             : <fulldecl>
                          | <symbol>

<declarationlist>         : <declaration>
                          | <declarationlist> ';' <declaration>

<declarations>            : DECLARATIONS <declarationlist> ';'

<declwithoptionalargs>    : <simpledec>
                          | <simpledec> <formalargs>

<definition>              : <declwithoptionalargs> IS <expression>

<definitionlist>          : <definition> ';'
                          | <definition> ';' <definitionlist>

<definitions>             : DEFINITIONS <definitionlist>

<delay>                   : DELAY UNTIL <expression> ';'
                          | DELAY WITH <listofexpressionsemi> UNTIL
                              <expression> ';'

<eff_exc_of>              : EFFECTS_OF
                          | EXCEPTIONS_OF

<effects>                 : EFFECTS <listofexpressionsemi>

<effectsmacro>            : <eff_exc_of> <call>

<exception>               : <expression> ';'
                          | <symbol> ':' <expression> ';'

<exceptions>              : EXCEPTIONS <exception>
                          | <exceptions> <exception>

<expression>              : <bexp1> '=>' <bexp1>
                          | <symbol> '=>' <bexp1>
                          | <bexp1> '=>' <symbol>
                          | <symbol> '=>' <symbol>
                          | <ifexpression>
                          | <typecaseexpression>
                          | <letexpression>
                          | <someexpression>
                          | <quantifiedexpression>
                          | <bexp1>
                          | <symbol>
```

```
<expressionlist>        : <expression>
                        | <expression> ',' <expressionlist>

<externalgroup>         : FROM <symbol> ':' <externalreflist>

<externalgrouplist>     : <externalgroup>
                        | <externalgroup> <externalgrouplist>

<externalref>           : <typedeclaration>
                        | <parameterdec>
                        | <functiondec>

<externalreflist>       : <externalref> ';'
                        | <externalref> ';' <externalreflist>

<externalrefs>          : EXTERNALREFS <externalgrouplist>

<fact>                  : <fact> <multop> <superfact>
                        | <symbol> <multop> <superfact>
                        | <fact> <multop> <symbol>
                        | <symbol> <multop> <symbol>
                        | <superfact>

<fafterass>             : <functions> END_MODULE
                        | END_MODULE

<fafterdecl>            : <parameters> <fafterparm>
                        | <fafterparm>

<fafterdefs>            : <externalrefs> <fafterext>
                        | <fafterext>

<fafterext>             : <assertions> <fafterass>
                        | <fafterass>

<fafterparm>            : <definitions> <fafterdefs>
                        | <fafterdefs>

<faftertype>            : <declarations> <fafterdecl>
                        | <fafterdecl>

<fieldvalue>            : <symbol> ':' <expression>

<fieldvaluelist>        : <fieldvalue>
                        | <fieldvalue> ',' <fieldvaluelist>

<formalargs>            : '(' ')'
                        | '(' ')' '[' <declarationlist> ']'
                        | '(' <declarationlist> ')'
                        | '(' <declarationlist> ')'
                            '[' <declarationlist> ']'
```

```
<fparagraphlist>        : <types> <faftertype>
                        | <faftertype>

<fulldecl>              : <simpledec>
                        | <fulldecl> ',' <symbol>

<functiondec>           : <ofunheader>
                        | <ovfunheader>
                        | <vfunheader>

<functions>             : FUNCTIONS <functionspeclist>

<functionspec>          : <vfunspec>
                        | <ofunspec>
                        | <ovfunspec>

<functionspeclist>      : <functionspec>
                        | <functionspec> <functionspeclist>

<ifexpression>          : IF <expression> THEN <expression>
                             ELSE <expression>

<init_deriv>            : INITIALLY
                        | DERIVATION

<interface>             : HIDDEN ';'
                        | <exceptions>

<invariants>            : INVARIANTS <listofexpressionsemi>

<letexpression>         : LET <qualificationlist> IN <expression>

<listofexpressionsemi>  : <expression> ';'
                        | <expression> ';' <listofexpressionsemi>

<localassertions>       : ASSERTIONS <listofexpressionsemi>

<localdefs>             : DEFINITIONS <definitionlist>

<logicalconstant>       : TRUE
                        | FALSE
                        | UNDEFINED
                        | '?'
                        | RESOURCE_ERROR

<mafterdecl>            : <parameters> <mafterparm>
                        | <mafterparm>

<mafterdefs>            : <externalrefs> <mafterext>
                        | <mafterext>

<mafterext>             : <invariants> <mafterinv>
```

```
                          | <mafterinv>

<mafterinv>               : <mappings> END_MAP
                          | <mappings> IMPLEMENTATIONS
                          | END_MAP
                          | IMPLEMENTATIONS

<mafterparm>              : <definitions> <mafterdefs>
                          | <mafterdefs>

<maftertype>              : <declarations> <mafterdecl>
                          | <mafterdecl>

<mapping>                 : <symbol> ':' <expression>
                          | <symbol> ':' <typespec1>
                          | <symbol> <formalargs> ':' <expression>

<mappinglist>             : <mapping> ';'
                          | <mapping> ';' <mappinglist>

<mappings>                : MAPPINGS <mappinglist>

<minusexp>                : '-' <minusexp>
                          | <term>
                          | '-' <symbol>

<mparagraphlist>          : <types> <maftertype>
                          | <maftertype>

<multop>                  : '*'
                          | '/'
                          | MOD
                          | INTER

<not>                     : NOT
                          | '~'

<ofunheader>              : OFUN <symbol> <formalargs>

<ofunsection>             : <localdefs>
                          | <delay>
                          | <exceptions>
                          | <localassertions>
                          | <effects>

<ofunspec>                : <ofunheader> ';'
                          | <ofunspec> <ofunsection>

<ovfunheader>             : OVFUN <symbol> <formalargs>
                              '->' <resultargs>

<ovfunspec>               : <ovfunheader> ';'
```

```
                          | <ovfunspec> <ofunsection>

<parameterdec>            : <declwithoptionalargs>
                          | <parameterdec> ',' <symbol>
                          | <parameterdec> ',' <symbol>
                              <formalargs>

<parameterdeclist>        : <parameterdec> ';'
                          | <parameterdec> ';' <parameterdeclist>

<parameters>              : PARAMETERS <parameterdeclist>

<predefinedtype>          : INTEGER
                          | REAL
                          | BOOLEAN
                          | CHAR

<qualifdeclist>           : <qualification>
                          | <simpledec>
                          | <symbol>
                          | <qualification> ';' <qualifdeclist>
                          | <simpledec> ';' <qualifdeclist>
                          | <symbol> ';' <qualifdeclist>

<qualification>           : <simpledec> '|' <expression>
                          | <simpledec> <relop> <expression>
                          | <symbol> '|' <expression>
                          | <symbol> <relop> <expression>

<qualificationlist>       : <qualification>
                          | <qualification> ';' <qualificationlist>

<quantifiedexpression>    : <quantifier> <qualifdeclist> ':'
                              <expression>

<quantifier>              : FORALL
                          | EXISTS

<range>                   : FOR <symbol> FROM <expression> TO
                              <expression>

<relop>                   : '='
                          | '~='
                          | '<'
                          | '>'
                          | '<='
                          | '>='
                          | INSET
                          | SUBSET

<resultargs>              : <declaration>
```

AD-A091 271 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK. VOLUME--ETC(U)
JUN 79 W L SUTTON, B A SILVERBERG, L ROBINSON N00123-76-C-0195
UNCLASSIFIED NOSC-TD-366-VOL-2 NL

2 of 2

END
DATE
FILMED
1-80
DTIC

```
<setexpression>          : '{' <expressionlist> '}'
                         | '{' '}'
                         | '{' <simpledec> '|' <expression> '}'
                         | '{' <symbol> '|' <expression> '}'
                         | '{' <expression> '..' <expression> '}'

<simpledec>              : <typespec1> <symbol>
                         | <setexpression> <symbol>
                         | <symbol> <symbol>

<someexpression>         : SOME <qualification>

<structureconstructor>   : STRUCT '(' ')'
                         | STRUCT '(' <expressionlist> ')'
                         | STRUCT '(' <fieldvaluelist> ')'

<structuretype>          : 'STRUCT\OF' '(' <declarationlist> ')'

<superfact>              : <minusexp> '^' <superfact>
                         | <symbol> '^' <superfact>
                         | <minusexp> '^' <symbol>
                         | <symbol> '^' <symbol>
                         | <minusexp>

<symbollist>             : <symbol>
                         | <symbol> ',' <symbollist>

<term>                   : <call>
                         | STRING
                         | NUMBER
                         | <logicalconstant>
                         | <term> '[' <expression> ']'
                         | <symbol> '[' <expression> ']'
                         | '(' <expression> ')'
                         | <term> '.' <symbol>
                         | <symbol> '.' <symbol>
                         | <unaryfunction> '(' <expression> ')'
                         | <setexpression>
                         | <vectorconstructor>
                         | <structureconstructor>
                         | <effectsmacro>

<typecaseexpression>     : TYPECASE <symbol> OF <caselist> END

<typeconstructor>        : SET_OF
                         | VECTOR_OF

<typedeclaration>        : <symbollist> ':' DESIGNATOR
                         | <symbollist> ':' <typespecification>

<typedeclarationlist>    : <typedeclaration> ';'
                         | <typedeclaration> ';'
```

```
                                <typedeclarationlist>

<types>                  : TYPES <typedeclarationlist>

<typespec>               : <typespec1>
                         | <setexpression>
                         | <symbol>

<typespec1>              : <predefinedtype>
                         | <structuretype>
                         | <unitedtype>
                         | <typeconstructor> <typespec>

<typespecification>      : <typespec>

<typespeclist>           : <typespec>
                         | <typespec> ',' <typespeclist>

<unaryfunction>          : CARDINALITY
                         | LENGTH
                         | NEW
                         | MAX
                         | MIN

<unitedtype>             : ONE_OF '(' <typespeclist> ')'

<value>                  : <init_deriv> <expression> ';'

<vectorconstructor>      : VECTOR '(' ')'
                         | VECTOR '(' <expressionlist> ')'
                         | VECTOR '(' <range> ':' <expression> ')'

<vfunheader>             : VFUN <symbol> <formalargs>
                             '->' <resultargs>

<vfunsection>            : <localdefs>
                         | <interface>
                         | <value>
                         | <localassertions>

<vfunspec>               : <vfunheader> ';'
                         | <vfunspec> <vfunsection>
```

```
<ilpl>                  : IMPLEMENTATION <symbol> IN_TERMS_OF
                             <symbollist> ';' <typesq> <parmsq>
                             <externalrefsq> <typemappingsq>
                             <initializationq> <implementationsq>
                             END_IMPLEMENTATION

<addop>                 : '+'

<aexpr1>                : <aexpr1> <addop> <aexpr2>
                        | <aexpr1> '-' <aexpr2>
                        | <aexpr2>

<aexpr2>                : <aexpr2> <multop> <aexpr3>
                        | <aexpr3>

<aexpr3>                : '-' <term>
                        | <term>

<bexpr0>                : <bexpr0> OR <bexpr1>
                        | <bexpr1>

<bexpr1>                : <bexpr1> AND <bexpr2>
                        | <bexpr2>

<bexpr2>                : NOT <bexpr3>
                        | <bexpr3>

<bexpr3>                : <aexpr1> <relop> <aexpr1>
                        | <aexpr1>

<body>                  : BEGIN <stmtgroup> END ';'

<caselist>              : TYPE <typespec> ':' <stmtgroup>
                        | <caselist> TYPE <typespec> ':'
                             <stmtgroup>

<declaration>           : <typespec> <symbollist>

<declarationlist>       : <declaration>
                        | <declarationlist> ';' <declaration>

<declarations>          : DECLARATIONS <declarationlist> ';'

<declarationsq>         : <declarations>
                        |

<do>                    : DO
                        | ASSERT

<elseclause>            : ELSE <stmtgroup>
                        | ELSIF <ifclause> <elseclause>
```

```
                        |

<event>                 : NORMAL
                        | RESOURCE_ERROR
                        | <symbol>

<eventcase>             : ON <eventlist> ':' <stmtgroup>

<eventcasep>            : <eventcase>
                        | <eventcasep> <eventcase>

<eventlist>             : <event>
                        | <eventlist> ',' <event>

<expect_until>          : EXPECTING
                        | UNTIL

<expr>                  : <bexpr0>
                        | <ifexpression>

<exprlist>              : <expr>
                        | <exprlist> ',' <expr>

<exprlistq>             : <exprlist>
                        |

<exprq>                 : <expr>
                        |

<externalgroup>         : FROM <symbol> ':' <externalreflist>

<externalgroupp>        : <externalgroup>
                        | <externalgroupp> <externalgroup>

<externalref>           : <typedec>
                        | <parameterdec>
                        | <functiondec>

<externalreflist>       : <externalref> ';'
                        | <externalreflist> <externalref> ';'

<externalrefs>          : EXTERNALREFS <externalgroupp>

<externalrefsq>         : <externalrefs>
                        |

<forhead>               : FOR <symbol> FROM <aexpr1> TO <aexpr1>
                        | FOR <symbol> FROM <aexpr1> BY <aexpr1>
                            TO <aexpr1>
                        | FOR <symbol> FROM <aexpr1>
                        | FOR <symbol> FROM <aexpr1> BY <aexpr1>
```

```
<formalargs>            : '(' ')'
                        | '(' ')' '[' <declarationlist> ']'
                        | '(' <declarationlist> ')'
                        | '(' <declarationlist> ')'
                            '[' <declarationlist> ']'

<formalargsq>           : <formalargs>
                        |

<functioncall>          : <symbol> '(' <exprlistq> ')'

<functiondec>           : <ofunheader>
                        | <ovfunheader>
                        | <vfunheader>

<header>                : <vhead> <symbol> <formalargs>
                            '->' <declaration> ';'
                        | <ohead> <symbol> <formalargs> ';'
                        | INITIALIZATION

<ifclause>              : <expr> THEN <stmtgroup>

<ifexpression>          : IF <expr> THEN <expr> ELSE <expr>

<implementations>       : IMPLEMENTATIONS <programp>

<implementationsq>      : <implementations>
                        |

<initialization>        : INITIALIZATION <declarationsq> <body>

<initializationq>       : <initialization>
                        |

<lhs>                   : <simplelhs>
                        | '(' <lhslist> ')'

<lhslist>               : <lhs>
                        | <lhslist> ',' <lhs>

<logicalconstant>       : TRUE
                        | FALSE

<multop>                : '*'
                        | '/'
                        | MOD

<ofunheader>            : OFUN <symbol> <formalargs>

<ohead>                 : OPROG
                        | OSUBR
```

```
<ovfunheader>          : OVFUN <symbol> <formalargs>
                             '->' <declaration>

<parameterdec>         : <simpledec> <formalargsq> <parmstar>

<parmdeclist>          : <parameterdec> ';'
                       | <parmdeclist> <parameterdec> ';'

<parms>                : PARAMETERS <parmdeclist>

<parmsq>               : <parms>
                       |

<parmstar>             : <parmstar> ',' <symbol> <formalargsq>
                       |

<predefinedtype>       : INTEGER
                       | REAL
                       | BOOLEAN
                       | CHAR

<program>              : <header> <declarationsq> <body>

<programp>             : <program>
                       | <programp> <program>

<raise_signal>         : RAISE
                       | SIGNAL

<range>                : FOR <symbol> FROM <expr> TO <expr>

<relop>                : '='
                       | '~='
                       | '<'
                       | '>'
                       | '<='
                       | '>='

<setbody>              : <exprlistq>
                       | <simpledec> '|' <expr>
                       | <expr> '..' <expr>

<setexpr>              : '{' <setbody> '}'

<simpledec>            : <typespec1> <symbol>
                       | <symbol> <symbol>

<simplelhs>            : <symbol>
                       | <simplelhs> '[' <expr> ']'
                       | <simplelhs> '.' <symbol>

<stmt>                 : <lhs> '<-' <expr>
```

```
                          | <functioncall>
                          | RETURN
                          | <raise_signal> '(' <event> ')'
                          | IF <ifclause> <elseclause> END_IF
                          | <forhead> <do> <stmtgroup> END
                          | <untilhead> <do> <stmtgroup> THEN
                              <eventcasep> END
                          | EXECUTE <functioncall> THEN
                              <eventcasep> END
                          | EXECUTE <symbol> '<-' <functioncall>
                               THEN <eventcasep> END
                          | TYPECASE <expr> OF <caselist> END

<stmtgroup>               : ';'
                          | <stmt> ';'
                          | <stmtgroup> <stmt> ';'

<structureconstructor>    : STRUCT '(' <exprlist> ')'

<symbollist>              : <symbol>
                          | <symbollist> ',' <symbol>

<term>                    : <symbol>
                          | NUMBER
                          | STRING
                          | <logicalconstant>
                          | <functioncall>
                          | '(' <expr> ')'
                          | <term> '.' <symbol>
                          | <term> '[' <expr> ']'
                          | <unaryfunction> '(' <expr> ')'
                          | <structureconstructor>
                          | <vectorconstructor>

<typedec>                 : <symbollist> ':' DESIGNATOR
                          | <symbollist> ':' <typespec>

<typedeclist>             : <typedec> ';'
                          | <typedeclist> <typedec> ';'

<typemappings>            : TYPE_MAPPINGS <typemappingsbody>

<typemappingsbody>        : <symbol> ':' <symbol> ';'
                          | <typemappingsbody> <symbol> ':'
                              <symbol> ';'

<typemappingsq>           : <typemappings>
                          |

<types>                   : TYPES <typedeclist>

<typespec>                : <typespec1>
```

```
                        | <symbol>
                        | <setexpr>

<typespec1>             : <predefinedtype>
                        | ONE_OF '(' <typespeclist> ')'
                        | STRUCT_OF '(' <declarationlist> ')'
                        | VECTOR_OF <typespec>
                        | SCALAR '(' <symbollist> ')'

<typespeclist>          : <typespec>
                        | <typespeclist> ',' <typespec>

<typesq>                : <types>
                        |

<unaryfunction>         : LENGTH
                        | INTPART
                        | FRACTPART

<untilhead>             : expect_until <eventlist>
                        | <forhead> expect_until <eventlist>

<vectorconstructor>     : VECTOR '(' <exprlistq> ')'
                        | VECTOR '(' <range> ':' <expr> ')'

<vfunheader>            : VFUN <symbol> <formalargs>
                              '->' <declaration>

<vhead>                 : VPROG
                        | VSUBR
                        | OVPROG
                        | OVSUBR
```

LMED

8