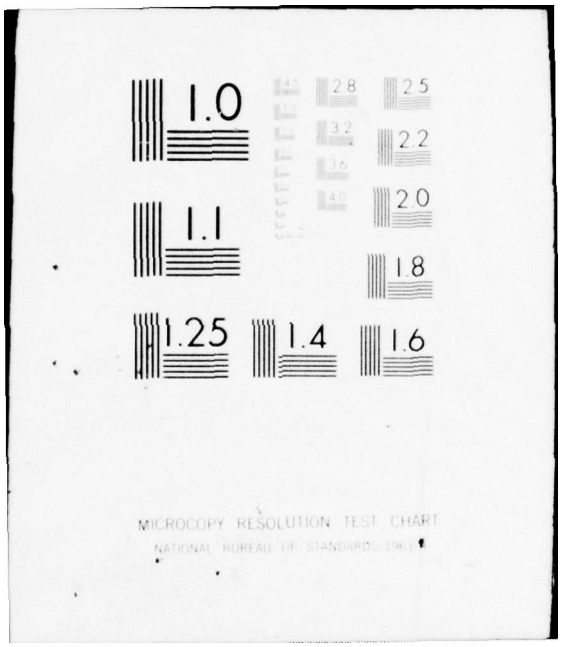
AD-A038 254 STANFORD RESEARCH INST MENLO PARK CALIF COMPUTER SCI--ETC F/G 9/2
SPECIAL REFERENCE MANUAL.(U)
JAN 77 L ROBINSON, O ROUBINE

UNCLASSIFIED CSL-45 N00123-76-C-0195 NL

1 OF 1
AD A038254

END

DATE FILMED
5-77


1.0
1.1
1.25
1.4
1.6
2.8
2.5
3.2
2.2
3.6
2.0
4.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963

12

Technical Report CSL 45

January 1977

SPECIAL REFERENCE MANUAL

3rd Edition

By: OLIVIER ROUBINE and LAWRENCE ROBINSON

Prepared for:

NAVAL ELECTRONICS LABORATORY CENTER
SAN DIEGO, CALIFORNIA 92152

CONTRACT N00123-76-C-0195

W. LINWOOD SUTTON, Contract Monitor

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

SPECIAL REFERENCE MANUAL

5. TYPE OF REPORT & PERIOD COVERED

Technical Report

6. PERFORMING ORG. REPORT NUMBER

CSL-45

7. AUTHOR(s)

Lawrence Robinson and Olivier Roubine

8. CONTRACT OR GRANT NUMBER(s)

Contract N00123-76-C-0195

9. PERFORMING ORGANIZATION NAME AND ADDRESS

Stanford Research Institute
333 Ravenswood Avenue
Menlo Park, CA 94025

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

Part of deliverable A006

11. CONTROLLING OFFICE NAME AND ADDRESS

Naval Electronics Laboratory Center
San Diego, CA 92151

12. REPORT DATE

November 1976

13. NO. OF PAGES

62 pages

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)

11 Jan 77

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited.

12 62p.

DDC
RECEIVED
APR 15 1977
C

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)

Formal specifications, Modularity, Abstract machines, Hierarchical structure, Logic, Language design

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

This document describes the specification language SPECIAL, which is a tool developed for the design of large software systems. The language is based on a methodology using the concept of a hierarchy of modules, and provides a convenient facility for the description of the properties of such modules. The syntax of the language is described, as well as the semantic notions related to its various constructs.

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

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

410 142

LB

TABLE OF CONTENTS

1. INTRODUCTION

The design and proof of large software systems can be facilitated by their decomposition into modules, as suggested by Parnas (Par 72 a, b). A design methodology based on this idea has been developed at SRI (Neu 74; Rob 75 a, b), and applied to the construction of several large systems, including a secure operating system (Neu 75). A significant aspect of this work is the design of SPECIAL, a SPECIfication and Assertion Language, which is described here.

The reader is assumed to be already familiar with the methodology itself. The concepts underlying the construction of the language are not emphasized here and will be the subject of a forthcoming document.

This manual is chiefly concerned with the syntax of the language and the associated semantic rules. A general presentation of the language and the objects it manipulates is first given. The "paragraphs" that may appear in a module specification are then described. The part of the language concerned with algebraic and arithmetic expressions is described in Section 10. Further sections contain the precedence rules for binary operators and the rules for referencing an object (scope rules).

SPECIAL has evolved through several forms. The present edition of this manual corresponds to the second version of the language.

2. GENERAL PRESENTATION OF THE LANGUAGE

The macroscopic unit expressed in SPECIAL is the specification of a module or of a set of mapping-function expressions. The first word of a specification, MODULE (1) in the former case, or MAP in the latter, provides the necessary distinction.

We are chiefly concerned with module specifications, because specifications of mapping-function expressions use only a subset of the syntax available for module specifications.

A module specification consists o sequence of paragraphs appearing between the header

MODULE <symbol>

---

(1) In the rest of this document, a word written in upper case refers to a SPECIAL r served word.

and the keyword END_MODULE. The paragraphs are entitled TYPES, DECLARATIONS, PARAMETERS, DEFINITIONS, EXTERNALREFS, ASSERTIONS (or INVARIANTS in the case of mapping functions), and FUNCTIONS (or MAPPINGS, for mapping function specifications) and must appear in that order. A paragraph starts with a keyword (e.g., TYPES) and ends at the beginning of the next paragraph or at the end of the module. If a paragraph is empty, it must be omitted; i.e., two paragraph titles may not follow each other.

Thus, the general outline of a module is something like

```
MODULE <symbol> (2)
    TYPES
      .
      .
      .
    DECLARATIONS
      .
      .
      .
    PARAMETERS
      .
      .
      .
    DEFINITIONS
      .
      .
      .
    EXTERNALREFS
      .
      .
      .
    ASSERTIONS
      .
      .
      .
    FUNCTIONS
      .
      .
      .
END_MODULE
```

The part of the language that deals with the objects of a module specification is known as the specification level; we describe later a more microscopic level known as the assertion level.

---

(2) Names appearing within angle brackets correspond to non-terminals of the grammar (see Appendix A); <symbol> is a particul ...n-terminal referring to any identifier that is not a reserved word; what is a legal identifier is described in Section 17.

3. THE OBJECTS USED IN THE LANGUAGE

The specification language manipulates several kinds of objects, all of which are "typed".

3.1. Types and Type Definitions

Each object manipulated by the language is associated with a type. For the descriptive purpose of this manual, it is sufficient to consider a type as a set of possible values.(3) An important aspect of the language is the description of the constraints imposed by types on arbitrary expressions. These constraints are the type rules of the language; they describe how operands of various types can be combined with particular operators to form an expression, and what type this expression will have.

The syntactic representation of a type is a type specification, which is a description of the properties of a type as a whole; type specifications are described in Section 5.3.

We distinguish various categories of types and describe them below:

Predefined types: The predefined types of the language are BOOLEAN, CHAR, INTEGER, and REAL.

Designator types: Designator types form a class of objects--designators--that are tokens for objects manipulated by the system being specified. Such objects may not be used as freely as, let us say, a number. All the objects with the same designator type are maintained by a particular module and can be created only by functions of this module (generally using the primitive NEW--see Section 12.15.).

Scalar types: Scalar types are types explicitly defined as sets of constants such that any two of them are disjoint (see the comments in Section 5.4.), e.g.,

```
direction : { left, right };
```

which defines "direction" as being a type, elements of which can have only one of the values "left" or "right"; those two symbols are implicitly declared as constants of the type "direction" by the type declaration. Note that such types are not ordered sets, and that only two operations are permitted on them: = and ~=. Objects of a scalar type also correspond to existing objects of the system but are not maintained by a particular module.

Constructed types: The language contains two unary type constructors, SET_OF and VECTOR_OF, used to define sets and vectors of objects of some given

---

(3) The term possible values should be distinguished from meaningful values, which are the values for which a particular assertion is true. In particular, the range of a function is a subset of the type of its result.

type, and two n-ary type constructors, ONE_OF and STRUCT, used to define united types and structured types, respectively. Since we view a type as a set of possible values, ONE_OF corresponds to the union operator on sets, and STRUCT corresponds to the Cartesian product. Note that type constructors apply to types, not to elements: in particular, a set type defined as "SET_OF t" is a type that represents a set of sets; i.e., each element of this type is a set of values of type t.

Subtypes: The notion of subtype corresponds to that of subset: the language provides the possibility of defining a type as an explicit set of constants of some existing type or as the set of the elements in a particular type that satisfy some property. The particularity of these types is that, for the purpose of type checking, they define only objects of the principal type, since it would require at least a theorem prover to verify the closure of a subtype under all the operations defined on the principal type.

The term primitive type refers to types that are sets of primitive values, one another disjoint; they are the predefined, designator, and scalar types. We can now define a type as being a primitive type, a subtype, or the result of the application of some of the type-constructors to a certain number of types.

3.2. Objects

There is a distinction between the objects manipulated by a module and those manipulated by the specification language, the former being a subset of the latter. The objects of a module are functions and parameters (designator and scalar types, although being families of objects rather than objects, are sometimes referred to as objects of the module, principally because they can be referenced in other modules). In addition, the language deals with so-called definitions, formal and result arguments, and locally bound variables. Each object has a name, a syntactic class, a scope, and a type. The syntactic class is either simple or functional; a simple object can be referenced by a single identifier (its name), whereas a functional object is referenced by an identifier followed by a list of "actual parameters" which are expressions of the language. The scope of an object is the part of a module specification where an object can be referenced after having been declared (the concept of declaration is described in the following section); for instance, the objects of a module (functions and parameters) have the entire module specification as their scope.

3.2.1. Functions

A function is an object of the module itself. Its syntactic class is always functional, and, in addition, a function has to belong to one of three categories. A category is either VFUN, OFUN, or OVFUN, corresponding to the V-, O-, or OV-functions of the methodology. Functions are either described in the module itself or referred to as external by appearing in the EXTERNALREFS paragraph. A function has a (possibly empty) argument list; every time the function is referenced, it must be followed by a list of expressions, the elements of which are in one-to-one correspondence with the formal arguments, the type of the elements being the same for each pair. In the case of a V- or OV-function, the function also has a "result argument" of a certain type. We

sometimes mention the type of a function, thereby meaning the type of its result argument.

3.2.2. Parameters

The parameters of a module are objects that receive a value at module initialization and cannot change thereafter. They can be either simple or functional objects, according to whether the value they receive is a simple value or a function (in the mathematical sense). They represent objects that are fixed in one particular module instantiation, but can be different in different instantiations. As already mentioned, parameters are objects of a module, and, as such, their scope is the entire module specification.

3.2.3. Definitions

A definition is a named object that can be textually replaced by a particular expression; it corresponds to what is generally known as a macro. Definitions can be either simple or functional, and they can be declared either in the DEFINITIONS paragraph, in which case their scope is the whole module specification, or in the DEFINITIONS section of a function, in the case of local definitions whose scope is the function specification.

Definitions differ from the general concept of macros in that the expression that is the body of the definition must be a valid expression when it is declared and not just when it is used (. .ules pertaining to the use of definitions guarantee the latter condition if the former one is satisfied). This precludes the use of free variables inside definitions. One of the reasons for this handling of definitions is that it generally prevents the same name from standing for two totally different meanings depending on where it is used.

3.2.4. Formal Arguments

These are the variables appearing in the argument list of a function specification. They may be used freely within the function specification. Formal arguments are always simple objects of the language.

3.2.5. Result Arguments

A V or OV-function must have one and only one result argument, a simple object that may be used in the function specification to refer to the value returned by that function.

3.2.6. Locally Bound Variables

Locally bound variables are objects whose binding remains in effect only within particular expressions, such as quantified expressions (see FORALL and EXISTS--Section 12.10.) or set and vector constructors (see Sections 12.7. and 12.8.). Another important use of such variables is in the so-called LET-expressions (see Section 12.13).

4. BINDING AND THE CONCEPT OF DECLARATION

An object of the language is a rather abstract entity; to be manipulated with a minimal degree of convenience, an object must be associated with a name. Note that it does not make any difference, as far as the properties of a function are concerned, whether its first argument is named "a" or "z", provided that the chosen name is used consistently. In addition, an object must have a type and always has a "scope". We are thus confronted with the problem of associating a name with a particular object (e.g., argument), and also of associating an object with a particular type. What one actually does is to associate a name with a type, and a name with an object, which thus receives the type associated with the name.

4.1. Simple and Multiple Declarations

The association of a name with a type is performed by a particular construct of the language known as a declaration. A simple declaration has the form:

```
<simple declaration> ::= <type specification> <symbol>
```

(the syntax for a typ specification is described in the next section). An extrapolation of this construct, referred to as a multiple declaration, can be used to associate several names with the same type specification, using the syntax:

```
<multiple declaration> ::= <simple declaration>
                         | <multiple declaration> ',' <symbol>
```

Generally, the association between a name and a particular object (the "binding") is performed implicitly by the place where the declaration of the name appears; for example:

```
VFUN f1( INTEGER i ) -> BOOLEAN b;
```

means that:

--The name i is associated with the type INTEGER, and is bound to be the first (formal) argument of f1.

--Similarly, the name b is associated with the type BOOLEAN and designates the result argument of f1.

--Lastly, f1 itself is declared as a V-function of the type (INTEGER -> BOOLEAN).

4.2. Global Declarations and Their Use

It is often the case in specifications that declarations are an important portion of the whole specification and that type specifications themselves can be quite complicated. With this in mind, SPECIAL provides the possibility of separating declarations and bindings (sometimes called the "deferred binding" facility), in the case of formal and result arguments and locally bound variables. This separation is achieved by declaring a name (i.e., associating it with a type) in a global DECLARATIONS paragraph (see Section 6), and performing the binding by letting the name alone appear where a declaration was expected. The rules for using deferred binding are described below:

When a declaration is expected, two cases are possible. If there is indeed a declaration, the name is associated with the type appearing in the declaration and is bound to the relevant object, as determined by the lexical position of the declaration; if the name has already appeared in a global declaration, the new local declaration supersedes the global one for the scope of the object. On the other hand, if a single name appears instead of the declaration, the name designates the corresponding object, the latter being associated with the type appearing in the global declaration of the name. It is obviously an error to use a single name instead of a declaration if the name has not appeared in the DECLARATIONS paragraph.

Remark: In the case of formal arguments to definitions and functions, the language allows a list of multiple declarations separated by ';'. However, if globally declared names are used instead of declarations, each name must be followed by a ';', and not by a ','; in other words, an argument list of the form

```
( INTEGER i, j; BOOLEAN b )
```

should be written, if i, j, and b had been declared globally

```
( i; j; b )
```

5. TYPES

The TYPES paragraph has two purposes: it allows the user to introduce the designators of the module and to associate names with type specifications (thus providing a macro-like facility for types). The names defined in the TYPES paragraph as particular type specifications are referred to as "named types".

5.1. Designators

A user-defined type, or designator, corresponds to a class of objects that are tokens for abstract objects implemented in the software system being specified. It must have a name, introduced by a declaration of the form:

<symbol> : DESIGNATOR;

The name introduced in this declaration will thereafter be a primitive type of its own.

5.2. Named Types

A type, as defined in Section 3.1., may involve an arbitrary number of type constructors, leading to a complex type specification. Such a type expression can be named in a statement like

<type name> : <type specification>;

<type name> is a new identifier that can textually replace the type specification appearing on the right-hand side of the equal sign.

5.3. Type Specification

A type specification is the syntactic representation of a type.

--A predefined type, a designator or a named type is represented by a symbol.

--A scalar type has a specification of the form:

<scalar type> ::= '{' <symbol> {',' <symbol>}* '}' (4)

--A subtype has the syntax of a set-expression (see Section 12.7)

--A structured type is defined as:

<structured type> ::= STRUCT '(' {<declaration> ';'}+ ')'

where <declaration> is

<declaration> ::= <simple declaration>
| <multiple declaration>

Each declaration introduces a name that is associated with a particular component of a structure, a "field".

--A united type specification is:

---

(4) The syntax descriptions used in this manual follow the extended BNF conventions described in Appendix A

```
<united type> ::= ONE OF '(' <type specification>
                        {',' <type specification>}* ')'
```

A united type is a peculiar concept: We defined have earlier (Section 3.1.) a primitive type as a set of simple values disjoint from all the other primitive types (in particular, for a purist, the predefined types should include INTEGER and "NON_INTEGER_REAL", with REAL being defined as ONE_OF(INTEGER,NON_INTEGER_REAL)). A united type which is the union of some primitive types is also a set of simple values but is not necessarily disjoint from the primitive types. In addition, the operations defined on the component types are not defined on the united type, with the exception of '=' and '~=' which are defined on any type. The use of objects of united types must abide by certain rules, described later in this document (see in particular TYPECASE--Section 12.12, and coercion rules--Section 14).

--Sets and Vectors: These are constructed types specified by statements of the form:

```
<constructed type> ::= {SET_OF | VECTOR_OF} <type specification>
```

The TYPES paragraph contains an arbitrary number of type declarations and might look like:

```
    TYPES

capability, unique_identifier : DESIGNATOR;
machine_word : ONE_OF(INTEGER, capability);
access_rights : VECTOR_OF BOOLEAN;
```

the syntax being:

```
<types paragraph> ::= TYPES {<type declaration> ';'}+

<type declaration> ::= <symbol> { ',' <symbol>}*
                       ':' <type specification>
```

5.4. Additional Comments

The following remarks concern the use of scalar types and subtypes; those may not appear explicitly except when used to define a named type, which means that any type specification having the syntax of a set expression may appear only inside the TYPES paragraph or the EXTERNALREFS paragraph, and within these paragraphs, may not be embedded in another type specification. In addition, some special rules apply to the use of scalar types, since some problems might arise in cases like:

```
    TYPES

traffic_light :  { green , yellow , red };
color : { red , yellow , green , cyan , blue , magenta };
```

Here, we have first to consider "green", "yellow", and "red" as three constants of type "traffic_light". Then, when seeing the second declaration, we must realize that "green", "yellow", and "red" are in fact constants of the type "color" and that "traffic_light" is a subtype of "color".

Thus, the definition of a scalar type is the following: A scalar type specification is a set of literal constants such that either none of them is ever used in any other type specification in the text, or some of them are used to define a subtype of the scalar type, i.e., they appear in a subset of the set_expression defining the scal type.

This definition precludes, in particular, the following pathological case: Assume that "color" is defined as above, but the user wishes to introduce a more sophisticated traffic light, to cope with a particularly dangerous intersection, as:

traffic_light: { green , yellow , red , left_arrow }

thus creating the problem of defining the type of each of the constants, since "traffic_light" can no longer be considered a subtype of "color". Is "green" of type "color" or of type "traffic_light", or is it "ONE_OF(color, traffic_light)"? Rather than defining complicated rules to solve this problem, we prefer to forbid the intermixing of constants of different types (the problem raised here can be solved by defining a scalar type that is a superset of both "color" and "traffic_light").

Some further thought should also be given to the concept of structured types. One has to realize that these types allow the manipulation of tuples of heterogeneous objects, but that their purpose is not to define complex data structures such as trees, as is ordinarily done in modern programming languages (e.g., Pascal). It is in fact one of the goals of the methodology, of which SPECIAL is only a part, to allow the modules themselves to support the specification of complex data structures, and there is no reason for those to be provided directly by the language. Therefore, recursively defined structured types are not allowed.

6. DECLARATIONS

The DECLARATIONS paragraph contains declarations (i.e., name/type associations) for names the binding of which is deferred, as described in Section 4. If the user does not wish to use the deferred binding facility, the DECLARATIONS paragraph can be omitted.

A declaration associates a name with a type and stays in effect within the entire module.

The syntax for the DECLARATIONS paragraph is:

```
<declarations> ::= DECLARATIONS {<declaration> ';'}+
```

with

```
<declaration> ::= <type specification>
                  <symbol> {',' <symbol>}*
```

Example:

```
    DECLARATIONS
BOOLEAN  b;
INTEGER   i,j;
capability   c, c1;
SET_OF capability  sc;
```

7. PARAMETERS

Starting with t _ keyword PARAMETERS, this paragraph contains declarations for all the module parameters (see Section 3.2.2). Since parameters of the module can optionally have arguments, the syntax for parameter declarations is similar to the syntax for global declarations, with the option of appending a formal argument list after each symbol:

```
<parameter declaration> ::= <type specification> <symbol>
                            [<formalargs>]
                            {',' <symbol>[<formalargs>]}* ';'

<formalargs> ::= '(' [<declaration> {';' <declaration>}*] ')'
```

8. DEFINITIONS

Definitions are syntactic shorthands for arbitrary expressio s. They are defined in the DEFINITIONS paragraph; that is, the name of a definition is associated with the expression for which it stands by a declaration of the form:

```
<definition> ::= <typespecification> <symbol> [<formalargs>]
                 IS <expression> ';'
```

The type specification must correspond to the type of the expression, this redundancy being considered beneficial to the clarity of specifications. The expression appearing in the definition may reference only the formal arguments, if any, and the objects that are bound globally, i.e., parameters, global definitions and functions.

The list of formal arguments may be omitted, or may be present but empty; there is a difference in that a definition with an empty argument list must be referenced as a functional expression throughout the specification (e.g., "foo()"), whereas a definition with no argument list must be referenced as a simple term (e.g., "fie"). There is no semantic difference between these two cases, the only difference being purely syntactic.

9. EXTERNALREFS

Although the decomposition of a software system into modules is intended to abolish the intermodule assumptions at the specification level, this is not always possible. A mechanism for referring to the functions, parameters, and designator or scalar types of other modules is therefore needed in order to describe these intermodule assumptions. This is the purpose of the EXTERNALREFS paragraph.

The paragraph itself is divided into as many groups as there are references to different modules. Each group starts with the header:

FROM <symbol> :

where <symbol> is the name of an external module, and the group contains a list of declarations and/or function headers.

A declaration has the regular declaration syntax, as in Section 6, or the syntax of a designator or scalar type declaration, as described in Section 5.

A function header is

```
{ VFUN | OFUN | OVFUN } <symbol> <formalargs>
            [ '[' <declaration> {';' <declaration>}* ']' ]
            [ '->' <declaration> ] ';'
```

If the first term is VFUN or OVFUN, then the result part (i.e., '->' <declaration>) should be present, and not for OFUN.

As will be shown, this syntax is also used to begin internal function specifications.

Example:

```
    EXTERNALREFS

FROM capabilities:
capability, unique_identifier: DESIGNATOR;
access_type: { read , write , execute , modify , append };
OVFUN  create_capability() -> c;
OVFUN  restrict_acces(capability c1,
                      VECTOR_OF access_type atv)
                        -> capability c;
 VECTOR_OF access_type atv;
VFUN  get_uid(c) -> unique_identifier u;

FROM segments:
OVFUN  create_segment(INTEGER i,
                      VECTOR OF capability s1v)
                     -> capability s;
VFUN  h_seg_exists(unique_identifier u) -> BOOLEAN b;
INTEGER  maxsegs;
```

10. ASSERTIONS

Assertions are constraints on the state of a module, imposed by the module specification.modules, imposed by a mapping. They can occur in two places: in the ASSERTIONS paragraph of the module specification (betwee EXTERNALREFS and FUNCTIONS paragraphs), or in the ASSERTIONS section of an individual function. The ASSERTIONS paragraph is used for specifying constraints on the parameters of the module, and aids in characterizing its initial state. Its syntax is:

<assertions> ::= ASSERTIONS {<expression> ';'}+

Consistent with the general rule, this paragraph is optional.

The ASSERTIONS section of a function is a list of conditions that must be guaranteed by the program calling the function. In many cases, these conditions concern the arguments to the function, e.g.,

```
OFUN restore_display (mem_addr);
EXCEPTIONS
  current_level < 1;
ASSERTIONS
  mem_addr MOD 6 = 0;
EFFECTS
   .
   .
   .
```

Assertions are different from exception conditions in that the latter must be detected by the program implementing a function, while assertions need not be checked by the implementing program. Thus, a function may fail to execute properly if a call is made that violates the assertions.

11. FUNCTIONS

The FUNCTIONS paragraph contains the specifications for all the functions of the module and is therefore its most important part.

The methodology deals with OFUN, OVFUN, and VFUN which all have their particularities, although also some common points.

A function definition consists of a header, possibly a set of local macros in a DEFINITIONS subsection, and other subsections which will be characterized for each class of function.

11.1. The Function Header

Similar to that described in Section 9, it looks like

```
{ VFUN ¦ OFUN ¦ OVFUN } <symbol> <formalargs>
          [ '[' <declaration> {';' <declaration>}* ']' ]
          [ '->' <declaration> ] ';'
```

As in definitions or external references, any of the formal arguments, as well as the result argument, may have been declared in the DECLARATIONS paragraph and thus not be preceded by a type specification.

11.2. Implicit Arguments

The optional list of declarations enclosed within square brackets in the function header is the list of "implicit arguments". An implicit argument to a function is an argument that should be provided by the system, and not by the user, when the function is called (in the actual implementation). In terms of specifications, any reference to a function that has been defined with implicit arguments should contain as many actual arguments as the sum of the number of formal and implicit arguments appearing in the function header.

Example:

A function whose header is:

```
VFUN f(INTEGER i)[process_id p] -> INTEGER j;
```

may be referenced as a function of two arguments, e.g.,

```
f(0, p1)
```

11.3. Local Definitions

By introducing a section with exactly the same syntax as the DEFINITIONS paragraph, the user has the possibility of defining macros that are known only inside one function specification. Such definitions have the advantage of being able to reference the arguments of the function that have only a restricted binding.

11.4. The ASSERTIONS section

The user may express relations between function arguments and other objects of the module, in a section that has the very same syntax as the ASSERTIONS paragraph, and a similar purpose.

11.5. The Subsections of a V-function

The two remaining sections of a V-function describe its formal properties. A V-function may be visible or hidden, and--independently--derived or primitive.

If a V-function is visible, it may be referenced in programs outside the module, so a list of exception conditions must appear in the function specification. This EXCEPTIONS subsection is similar to the one appearing in O- and OV-function specifications and is detailed in Section 11.6.

On the other hand, a hidden V-function may not be called outside the module and needs no EXCEPTIONS subsection. In such a case, the reserved word HIDDEN appears in its place.

The last subsection of a V-function specification can either define the initial value of the V-function (for primitive V-functions) or explicitly describe the functions value in terms of other objects of the module (for derived V-functions). The former case is described through an assertion of the form

INITIALLY <expression> ';'

where <expression> is an arbitrary Boolean expression. In the latter case, the syntax is

```
DERIVATION <expression> ';'
```

Here, the expression must have the type of the result argument of the function.

comments:

--The set of all primitive V-functions entirely defines the current state of the machine described by the module. The set of their initial values thus defines the initial state of the module.

--Derived V-functions are often used to restrict or to package the information visible outside the module.

--Note that a V-function that is both hidden and derived is very similar to a global definition.

The general syntax for a V-function specification is thus

```
VFUN <symbol> ([<declaration> {',' <declaration>}* ])
                                  -> <declaration> ';'
[DEFINITIONS { <definition> ';' }+ ]
{HIDDEN ';' | [ EXCEPTIONS { <expression> ';'}+ ]}
[ ASSERTIONS { <expression> ';' }+ ]
{INITIALLY | DERIVATION} <expression> ';'
```

11.6. The Subsections of O- and OV-functions

As opposed to V-functions, O- and OV-functions are always visible and consequently may have an EXCEPTIONS subsection. Since both perform some operations that affect the state of the module, they also have an EFFECTS subsection.

11.6.1. The EXCEPTIONS Subsection

This subsection contains a list of conditions that, if any is satisfied (i.e., evaluates to TRUE when the function is called), should prevent any effect from taking place and/or any value from being returned. If an exception condition is raised, control is returned to the calling program with "appropriate" notification to the caller. Exactly how this notification is made depends on the conventions of the programming language; we mention here only the constraints imposed by the specification language.

An exception condition is a Boolean expression; the order in which the conditions are listed is important: The meaning of the EXCEPTIONS subsection is that each condition should be checked in turn, and only if all the conditions are false should the effects take place and/or the value be returned. The calling program may depend on the order in which the exception conditions are checked. For example, a reasonable EXCEPTIONS subsection might be:

```
    EXCEPTIONS
h_interrupt_set(i) = FALSE;
h_int_handler_offset(i) = ?;
```

(where the two functional expressions are likely to be V-function references), which might mean that i does not refer to any interrupt in the first exception and that, if it indeed does, no interrupt handler exists for it in the second exception; this illustrates that the second exception might be meaningless had not the first one been checked before.

There is also a particular construct intended to be applied only in the EXCEPTIONS subsection:

```
EXCEPTIONS_OF <call> ';'
```

where <call> should be a reference to an external function. The interpretation of this construct is that for each exception condition of the external function, there exists a logically equivalent exception condition in the current function. The order in which the exceptions appear in the external function is preserved, and the corresponding conditions are to be checked after all those that precede the "EXCEPTIONS_OF" expression.

The syntax for the EXCEPTIONS subsection can be described as

```
<exceptions> ::= EXCEPTIONS { <expression> ';'
                            | EXCEPTIONS_OF <call> ';' }+
```

Note that this expression forces at least one expression to occur in the EXCEPTIONS subsection; if the function has no exception, the whole subsection should be omitted.

11.6.2. The EFFECTS Subsection

The EFFECTS subsection contains a list of assertions that are Boolean expressions and have the following meaning: If the function is referenced and no exception is detected, then, after the function call, the conjunction of all the assertions appearing in the EFFECTS subsection may be assumed to be TRUE.

If the EFFECTS subsection of function f contains the assertions a1, a2,...,an, i.e., looks like

```
EFFECTS a1; a2;...; an;
```

and the EXCEPTIONS subsection refers to e1,...,em (i.e., EXCEPTIONS e1; e2;...;em;) then, using Hoare's notation, a reference to the function f has the meaning

```
NOT(e1 OR e2 OR...OR em) {f} a1 AND a2 AND...AND an
```

Note that, in the EFFECTS subsection, the order of the assertions is irrelevant.

In the case of an OV-function, one of the assertions might state the equality of the result argument with some expression, thus defining the value returned by the function, but this is not mandatory; the result argument should, however, appear in at least one of the effects, in order not to be undefined.

11.6.3. The DELAY Conditions

In addition to the EXCEPTIONS and EFFECTS subsections, an O- or OV-function may contain an arbitrary number (although generally only one) of so-called DELAY expressions which have the syntax:

```
DELAY [ WITH <expression> ';' {<expression> ';'}* ]
      UNTIL <expression> ';'
```

The DELAY section of a function allows the specification of a condition that must be guaranteed when the function is executed (just as in the ASSERTIONS and EXCEPTIONS sections), such that the execution of the function is delayed if the condition is not satisfied. This construct is used in synchronization primitives and other multi-process applications.

The "UNTIL" part specifies the condition that must be guaranteed when the function is executed. The optional "WITH" part specifies some state change that is to occur if the condition is not satisfied. Note that the exception conditions apply to the state after the "WITH" effects, when those are executed. For instance, the expression:

```
DELAY WITH 'process_state(p) = blocked;
           EFFECTS_OF enqueue(sem, p);
      UNTIL value(sem) > 0;
```

might appear in the specification of a synchronization primitive.

This concludes our description of the macroscopic or specification level of the language. The microscopic (or assertion) level deals with the kind of construct we have been so far referring to as <expression> or assertion, with the distinction that an assertion is an expression of type BOOLEAN. This part of the language is described next.

12. EXPRESSIONS

The assertion language has a relatively large set of syntax rules for writing concise mathematical expressions to describe conditions related to a function behavior. In certain cases, the semantics associated with these rules may require a detailed description. This section explores the different constructs used in mathematical expressions; an interesting aspect to be considered is the set of type rules associated with each operator.

12.1. Atomic Expressions

These are the simplest kind of expression (from the syntactic standpoint); an atomic expression can be one of the following:

--Numeric constant: A numeric constant may be either of type INTEGER (any nonempty sequence of digits), or of type REAL (a string of digits followed by a period followed by another string of digits followed by the letter E followed by an optionally signed string of digits, where either the integer part or the fractional part may be omitted, and the decimal point the exponent may also be omitted--but not both).

--Character and string constant: A constant of type CHAR is a single, printable ASCII character enclosed within two ` (back apostrophe). A string constant has the type "VECTOR_OF CHAR" and consists of any sequence of printable ASCII characters enclosed within two " (double-quote). The precise rules concerning character and string constants may depend on the implementation (see Section 17).

--Boolean constant: These are the keywords TRUE and FALSE.

--Symbolic constant: Any identifier appearing in a scalar type declaration will subsequently be considered a constant of that type. In addition, the two symbols UNDEFINED and ? are two synonyms which, for each type, refer to a particular value of that type, different from any other value (the particular type is determined by the context where they appear). It is a part of the verification task to check that the implementation of any visible V- or OV-function can never return the value UNDEFINED.

--Reference to an object: The name of any object that has a binding and whose syntactic class is not functional is an atomic expression whose type is that of the object. Such references can be to parameters or definitions when they have been declared without arguments, formal arguments of functions and definitions, result arguments of V- and OV-functions, and simple variables within expressions where they have a binding (i.e., within set and vector constructors, quantified expressions, and expressions starting with LET or SOME).

12.2. V-function References

A reference to a V-function has the form

['] <symbol> <actual list>

where <actual list> is defined as

'(' [<expression> {',' <expression>}*] ')'

and is used as such in the rest of this document. It corresponds to the actual arguments, which must match in number and type the formal arguments of the function definition.

<symbol> should be the name of a V-function either defined in the module or appearing in the EXTERNALREFS paragraph.

The first symbol, ' , is optional. Its meaning, within the EFFECTS part of a function specification, is that the reference is made to the value of the V-function immediately after all the effects of the O- or OV-function being specified have occurred; if it is omitted, the reference is to the value immediately before the call. Note that this is meaningful only in the EFFECTS section, since in any other place there is no distinction between before and after, and in the EXCEPTIONS subsection, a reference is always made before any effect takes place.(5)

The type of a V-function reference is the type of the function itself.

12.3. References to Functional Objects

We deal here with parameters and definitions introduced with an argument list. A reference to such objects has a syntax very similar to that of a V-function reference:

<symbol> <actual list>

with, for definitions, the usual meaning of a macro expansion, the type being that of the macro, and, in the case of a parameter, that of a mere reference to its value.

12.4. O- and OV-function References

A reference to an O- or OV-function is made by the use of the keyword EFFECTS_OF

EFFECTS-OF <symbol> <actual list>

In the case of O-functions, this expression is of type Boolean; its value is UNDEFINED if some exception is detected; it is FALSE if the effects of the O-function that is referenced are never satisfiable. A value of TRUE, on the contrary, means that all the assertions specified in the EFFECTS section of the corresponding O-function hold.

In the case of an OV-function, the type of the expression is that of the result argument of the function, and the expression itself is both a reference to the value of the OV-function and an indication that all its effects are true at the point where the EFFECTS_OF appears.

The next sections describe the operators used in the language. They are classified by the types on which they operate; the word "mode" appearing in a type description refers to a particular type which does not have to be

---

(5) We must add at this point an important historic note: Until now, all the module specifications derived from Parnas' ideas used quoted expressions to refer to old values of V-functions. However, this notation could not be used consistently because in the same function specification, unquoted references would have two different meanings, depending whether they appeared in the exceptions or in the effects of the function. Permuting the convention takes care of this possible confusion.

specified. The discussion concerning precedence of various operators is, however, deferred to Section 12.

12.5. Boolean-Valued Operators

12.5.1: Unary Operator

NOT

Syntax: { NOT ¦ ~ } <expression>
Type: BOOLEAN --> BOOLEAN
Meaning: logical negation of operand

12.5.2: Binary Operators

AND and OR

Syntax: <expression> {AND ¦ OR} <expression>
Type: BOOLEAN X BOOLEAN --> BOOLEAN
Meaning: usual conjunction or disjunction; *error* if one of the operands is not Boolean.(6)

=>

Syntax: expression_1 => expression_2
Type: BOOLEAN X BOOLEAN --> BOOLEAN
Meaning: usual implication, i.e.,
(expression_2 OR NOT expression_1) = TRUE
error if one of the expressions is not Boolean.

12.5.3: Relational Operators

=, ~=

Syntax: exp_1 {= ¦ ~=} exp_2
Type: mode X mode --> BOOLEAN
the types of exp_1 and exp_2 may be either:
any type, provided it is the same for both,(7)
or: any type for one expression and UNDEFINED for the other,
or: UNDEFINED for both;
Meaning: usual = or ≠

<, <=, >=, >

Syntax: exp_1 { > ¦ >= ¦ <= ¦ < } exp_2
Type: number X number --> BOOLEAN
(where number is either INTEGER or REAL)
Meaning; the usual order relations on numbers.

---

(6) When the term *error* is used, it does not refer to any particular value, but rather means that under the conditions associated with it a specification would be incorrect, as far as SPECIAL is concerned. In other terms, it is considered as erroneous to say "1 AND TRUE" as to say "DO I = 1, 10" in a specification.

(7) For more about type matching rules, see Section 14.

12.6. Operations on Numbers

The operations on numbers are the usual +, -, *, ^, and /, where - can be either unary or binary, and ^ is the exponentiation operator; in addition, the operator MOD is also provided. The meaning and use do not depart from those generally encountered. Numbers are either of type INTEGER or of type REAL. Both types can be intermixed according to the following rules: A binary operation involving two integers will have the type INTEGER, and an operation where at least one of the operands is real will have the type REAL. In particular, the operator /, when applied to two integers, will yield the integer part of the quotient of the two operands, whereas if one of the operands is of type REAL, the result should be the quotient itself. In addition, some primitives are provided in the language for specifying the integer and fractional parts of a real (see Section 12.16.). The use of MOD is restricted to INTEGER operands.

All the binary operators on numbers, as well as those on Booleans, or on sets are left-associative, except ^ which is right-associative.

12.7. Operations on Sets

12.7.1. Unary Operator

CARDINALITY

Syntax: CARDINALITY(<set expression>)
Type: set-mode --> INTEGER
Value: The number of elements in the set designated by its argument

12.7.2. Binary Operators

UNION

Syntax: <set expression> UNION <set expression>
Type: SET_OF m1 X SET_OF m2 --> SET_OF m3
where m3 is the larger of m1 and m2 if they match, and their union otherwise; note that, when the types of the two arguments do not match, the result will have the type of the minimal union of the two types, e.g.,

If the type of s1 is "SET_OF INTEGER"
and the type of s2 is "ONE_OF(SET_OF BOOLEAN, SET_OF CHAR)"
then s1 UNION s2 will be a
"ONE_OF(SET_OF ONE_OF(BOOLEAN, INTEGER),
SET_OF ONE_OF(CHAR , INTEGER))"

Meaning: The usual union of two sets.

INTER

Syntax: <set expression> INTER <set expression>
Type: SET_OF m1 X SET_OF m2 --> SET_OF m3
where m3 = m1 if m1 is the same as m2; generates an error if m1 and m2 refer to disjoint types, and m3 is the intersection of m1 and m2 otherwise.
Example:

If s1 is of type "SET_OF ONE_OF(INTEGER, BOOLEAN)"
and s2 of type "ONE_OF(SET_OF INTEGER, SET_OF CHAR)",
then s1 INTER s2 is of type "SET_OF INTEGER"

Meaning: usual set intersection.

DIFF

Syntax: set_1 DIFF set_2
Type: SET_OF m1 X SET_OF m2 --> SET_OF m1
Generates an error if m1 and m2 cannot have any common value. Example:

If set_1 is of type "SET_OF ONE_OF(INTEGER, BOOLEAN)" and set_2 is of type "ONE_OF(SET_OF INTEGER, SET_OF CHAR)"), then the result will have the type of the first argument.

Meaning; {x | x ∈ set_1 AND x ∉ set_2}

12.7.3. Predicates

INSET

Syntax: <expression> INSET <set_expression>
Type: m1 X SET_OF m2 --> BOOLEAN
Generates an error if m1 and m2 are totally disjoint types (as with DIFF).
Meaning: The expression is TRUE if the first operand is in the set represented by the second.

SUBSET

Syntax: set_1 SUBSET set_2
Type: SET_OF m1 X SET_OF m2 --> BOOLEAN
m1 and m2 are subject to the same constraints as with INSET.
Meaning: The expression is TRUE iff set_1 is a subset of set_2. Note that "s1 SUBSET s2" can be defined as :
FORALL x INSET s1: x INSET s2

12.7.4. Set Constructors

Explicit Constructor

Syntax: '{' expression_1,..., expression_n '}'
(Here, the curly brackets are actual symbols of the language.)
Type: If all the expressions have the same type, let us say t, then the result will be of type "SET_OF t"; otherwise, it will be "SET_OF ONE_OF(t1,..,tj)" where the united-type is the minimal union of the types of the expressions.
Value: The set S such that:
expression_i ∈ S (i = 1, 2,..,n)

Implicit Constructor

Syntax: '{' <simple declaration> '|' <expression> '}'
Type: <expression> must be Boolean; result is of type "SET_OF t" where t is the type of the variable appearing in the declaration.
Value: "{t x | e(x)}" is the set of all x's of type t such that e(x) is TRUE.

Interval Constructor

Syntax: '{' <expression1> '..' <expression2> '}'
Type: Both expressions must have the type INTEGER or REAL (or the union of both), and the result is a set of INTEGER, if both expressions were of type INTEGER, a set of REAL otherwise.
Value: { e1 .. e2 } is the set of numbers i such that e1<=i<=e2 (note that this expression will be a set of INTEGER or REAL, depending on the type of e1 and e2).

12.8. Operations on Vectors

Extractor

Syntax: <vector expression> [<integer expression>]
Type: If the first operand is of type "VECTOR_OF t", the result will be of type t.
Value: v[i] is the ith component of the vector v.

LENGTH

Syntax: LENGTH(<vector expression>)
Type: VECTOR_OF mode --> INTEGER
Value: the number of components (dimensionality) of its operand.

Explicit Constructor

syntax: VECTOR(exp_1,...,exp_n)
Type: If exp_1,...,exp_n are all expressions of type t, then the result will be of type "VECTOR_OF t"; if the types are different, the result will be a vector of the minimal union of these types.
Value: V such that
LENGTH (V) = n and
V[i] = exp_i for i = 1, 2,..., n

Implicit Constructor

Syntax: VECTOR '(' FOR <symbol> FROM <expression>
TO <expression> ':' <expression> ')'
Type: If the last expression always has the type t, then the result will be a "VECTOR_OF t" (the expressions following FROM and TO must have the type INTEGER). Note that this is the only place in the language where a name is bound to an object that need not be previously declared; the symbol following FOR is a locally bound variable whose type is always INTEGER.
Value: V = VECTOR(FOR i FROM exp1 TO exp2 : f(i))
is the vector such that:
LENGTH(V) = exp2 - exp1 + 1 and
V[i] = f(exp1 + i - 1) for i = 1, 2,...,(exp2 - exp1 + 1)

12.9. Operations on Structures

Extractor

A reference to a particular field of a structure-expression (i.e., an expression whose type is a structured type) can be made by appending a '.' followed by the name of the particular field after the structure-expression itself. The syntax is thus:
<expression> '.' <symbol>

Type: The type is that associated with the field name in the structure declaration.
Meaning: The structure expression being a tuple, the extractor refers to the value of the nth element of this tuple, if the field name appeared in the nth position in the structure declaration.

Explicit Constructor
Syntax: '<' exp_1,...,exp_n '>'
Type: t1 X .... X tn
where ti is the type of the ith expression in the list.
Value: the tuple whose ith element is exp_i, for i from 1 to n.

Implicit Constructor
Syntax: '<' FOR <symbol> FROM <expression> TO <expression>
':' <expression> '>'
Type: t X .... X t
i.e., the n-fold Cartesian product of the type of the third expression, where n is the difference between the second and first expressions.
Meaning: The tuple < exp_1,...,exp_n > where exp_i is the value of the third expression evaluated when the indicial variable (i.e., the <symbol>) has the value of the expression following FROM, augmented of i-1.

12.10. Quantified Expressions

Although quantified expressions are constrained to yield a Boolean value, they have been placed in this separate section because of their particular syntax rules.

Universal Quantifier
Syntax: FORALL {<qualification> ¦ <declaration>}
{ ';' {<qualification> ¦ <declaration>} }*
':' <expression>

with: <qualification> ::= {<simple declaration> ¦ <symbol>}
{ '=' ¦ '~=' ¦ '>' ¦ '>=' ¦ '<=' ¦ '<'
¦ SUBSET ¦ INSET ¦ '¦'} <expression>
Type: BOOLEAN
Meaning: as in predicate calculus
Remarks: The <expression> following the ':' will presumably depend on the variables declared after the quantifier. These variables are bound locally, only in the last expression (see Scope Rules, Section 12). The optional qualification may be used to further qualify the variable, without complicating the main quantified expression.
The general way of expressing something such as "for all x's such that P(x), Q(x) is true" is:

```
FORALL x ¦ P(x) : Q(x)
```

whereas "for all x in the set S, Q(x) is true" would be written as:

```
FORALL x INSET S : Q(x)
```

(Note that, in the two examples above, we used the deferred binding facility, presuming that x appeared in the DECLARATIONS paragraph.)
Several variables may be quantified simultaneously, e.g.,

```
FORALL x ¦ P(x) ; y ¦ Q(y) ; z : R(x, y, z)
```

In such a case, each quantified variable may appear in its own qualification and in the main (quantified) expression, but not in the qualification of another variable.
The above expression has the following meaning:

```
FORALL x : FORALL y : FORALL z : P(x) AND Q(y) => R(x, y, z)
```

Existential Quantifier

Syntax:

```
EXISTS {<qualification> ¦ <declaration>}
       { ';' {<qualification> ¦ <declaration>} }*
       : <expression>
```

Type: BOOLEAN
Meaning: as in predicate calculus
Remarks: same as above.

12.11. Conditional Expressions

Syntax: IF <expression> THEN <expression> ELSE <expression>
Type: Consider the expression

```
IF P THEN e1 ELSE e2
```

If e1 and e2 have the same type, that type will be the type of the whole expression; if one of e1 and e2 (or both) is UNDEFINED or ?, then the type will match anything. If the types of e1 and e2 do not match, then the type of the expression will be the union of the type of e1 and the type of e2.
Meaning: x = IF P THEN e1 ELSE e2 is an abbreviation for
(P AND (x = e1)) OR ((NOT P) AND (x = e2)
Remark: The ELSE part must always be present!

12.12. TYPECASE Expressions

This kind of expression is probably the most complex construct of the language. It is intended to be used in cases where the type of an object used in the system being specified cannot be known before execution time. Its purpose is to allow a temporary alteration of the type of an object, from a united type to one of the component types. This is particularly desirable because it permits the application to the object of some operations that would not be defined on the united type without compromising the safety of type-checking.

Syntax:

```
TYPECASE <symbol>  OF
type_1:  exp_1;
type_2:  exp_2;
      .
      .
      .
type_n:  exp_n;
END
```

Type: The type of the whole expression will be the minimal union of the types of exp_1,..., exp_n, in the sense that, if exp_1,.. exp_n all have the same type t, then t will be the type of the expression, otherwise, the type will be ONE_OF(t1, t2,...,tk) where t1,..,tk are all different, and are types of exp_1,...,exp_n (in no particular order).
Meaning: The value of the expression is more easily understood by using the meta-function "type_of" the value of which would be the type of its single argument.
Then, for instance

```
a = TYPECASE w of
type_1:  exp_1;
type_2:  exp_2;
type_3:  exp_3;
END;
```

would mean

```
((type_of(w) = type_1) AND a = exp_1)
OR ((type_of(w) ~= type_1
      AND type_of(w) = type_2)
    AND a = exp_2)
OR ((type_of(w) ~= type_1
       AND type_of(w) ~= type_2
       AND type_of(w) = type_3)
    AND a = exp_3)
```

Remarks: There are several restrictions imposed on the use of this construct. We will call

--case variable the variable following the keyword TYPECASE;
--case labels the types type_1,..., type_n;
--case expressions exp_1,..., exp_n;

--The type of the case variable must be a united type equal to the type that would be obtained by taking the union of all the case labels.

Example: If w is of type

```
ONE_OF(INTEGER, BOOLEAN, capability)
```

then

```
TYPECASE w of
INTEGER:  w;
BOOLEAN:  IF w = TRUE THEN 1 ELSE 0;
capability:  get_uid(w);
END;
```

is a valid expression;

```
TYPECASE w of
ONE_OF(INTEGER, BOOLEAN): TRUE;
capability:  FALSE;
END;
```

is also valid.

(As an illustration, let us mention that the first expression would have the type ONE_OF(INTEGER, "type_of get_uid"), whereas the second one would have the type BOOLEAN.)

On the contrary,

```
TYPECASE w OF
INTEGER: 1;
BOOLEAN : 0;
END;
```

is incorrect because one of the types is missing.

--Since the expression itself has a type, regular type checking is performed.

--When the type of each case expression is evaluated, if the case variable appears in the expression, it is taken to have the type of the corresponding case label.

12.13. LET Expressions

It is sometimes necessary to bind some variables locally inside a particular expressions. Yet their use is generally very limited. When it seems to be really necessary, the user can declare a list of variables immediately before using them; their scope is restricted to one single expression, and the syntax for this construct is:

```
LET <qualification> { ';' <qualification> }*
        IN <expression>
```

The value and the type of this expression are those of the expression following the keyword IN. The closest approximation to the semantics of:

```
y = LET x ¦ P(x) IN Q(x)
```

is something like

∃ x such that P(x) AND (y = Q(x))

but this is only an example of the use of this construct.

12.14. SOME Expressions

The SOME construct can be viewed as a kind of set extractor. The syntax is:

SOME <simple declaration> {INSET ¦ '¦'} <expression>

The meaning of an expression such as

SOME x INSET S

is: LET x INSET S IN x

and similarly if '¦' is used.

The following example illustrates the use of this construct:

```
EFFECTS_OF wake_up( SOME process p
                        INSET waiting_processes(sem1))
```

which might be an assertion inside the specification for a synchronization primitive.

12.15. The NEW Primitive

NEW is a special construct that has the same syntax as a function of one argument:

NEW '(' <symbol> ')'

where the argument should be the name of a designator type. The type of this expression is precisely this identifier, and the meaning is that NEW(t) represents an object of type t that has never been used before.

An error will occur if the argument is not a DESIGNATOR (even an external DESIGNATOR will not work).

12.16. Miscellaneous Operators

In addition to the various operators and constructs presented above, the language contains several functional primitives.

MAX, MIN

Syntax: {MAX ¦ MIN} '(' <expression> ')'
Type: SET_OF number --> number
(where number is either INTEGER or REAL)
Value: the largest (respectively smallest) element in the set described in the expression.

SUM

Syntax: SUM '(' <expression> ')'
Type: SET_OF number --> number
or: VECTOR_OF number --> number
Value: the arithmetic sum of all the elements in the set (respectively vector).

INTPART, FRACTPART

Syntax: {INTPART | FRACTPART} '(' <expression> ')'
Type: REAL --> INTEGER (resp. REAL)
Value: These two constructs can be viewed as predefined macros:
INTEGER INTPART(REAL x) IS SOME INTEGER y | y <= x AND y+1 > x;
REAL FRACTPART(REAL x) IS x - INTPART(x);
They define, respectively, the integer and fractional part of their argument (surprised?).

LOG

Syntax: LOG '(' <expression1> ',' <expression2> ')'
Type: number X number --> REAL
Value: the logarithm of <expression2> in the base <expression1>.

******

This concludes our syntactic description of the language for module specifications. We will now concern ourselves with some additional precisions concerning the use of certain objects.

13. PRECEDENCE

The large number of operators in the assertion language requires a precise definition of the precedence ordering, that is, if op1 and op2 are two operators (let us say binary), we say that op1 has a higher precedence than op2 if

e1 op1 e2 op2 e3

has the same meaning as

(e1 op1 e2) op2 e3

and

e1 op2 e2 op1 e3

has the same meaning as

e1 op2 (e2 op1 e3)

(Note that any expression can always be enclosed within parentheses.)

A general rule is that an operator returning a Boolean valued expression has always a lower precedence than an operator that returns another type. This rule is of course insufficient to compare operators returning the same type of expression.

The ordering is partial, since the operators cannot be freely combined in the text (e.g., INTER can hardly be compared with +).

The operators are listed in decreasing order of precedence. Constructs that do not lead to ambiguities (reference to functions, LENGTH, CARDINALITY, NEW, vector subscripting, TYPECASE, FORALL, EXISTS, vector and set constructors, LET, and SOME) are not listed, but it should be noted that, when used as operands with any of the operators listed below, the constructs using IF, LET, SOME, FORALL, EXISTS, and TYPECASE should be enclosed within parentheses.

INTEGER	BOOLEAN	sets
unary -		
*, /, MOD		INTER
+, -		UNION, DIFF
	=, ~=, <, <=, >=, >,	
	INSET. SUBSET	
	NOT	
	AND	
	OR	
	=>	

Of interest are the places of NOT, INSET and => : thus

```
(NOT  x) INSET  S
```

is meaningful only if x has the type BOOLEAN and s is a set of BOOLEAN (for instance {TRUE})

```
NOT x INSET s  =>  P(x)
```

means (x INSET s) OR P(x) .

14. SCOPE RULES

A name designates an object by two mappings: one from the name to a type and the other from the name to a "binding" (in the sense described in Section 4).

The basic rule is that a name cannot be used in an expression if the value of either of the two mappings is undefined.

The dual possibility of associating a name with a type through a declaration has been described at length in Section 4 and is not repeated here.

14.1. Establishing the Binding

The binding of an object is established by declaring the name one wishes to associate with the object in a predetermined place in the text, which depends on the category of the object itself (by "declaring", we mean that the name can appear in a declaration, in a header, or in the place of a declaration in the case of deferred binding).

Here is a table that indicates the place where a binding is established.

Object	Binding
Global definition	Definition in the DEFINITIONS paragraph
Local definition	Definition in the DEFINITIONS section of a function
Formal argument	When appearing in a function or macro header
Result argument	"
Quantified variable	When appearing after FORALL and EXISTS
Locally bound variable	When appearing between '{' and '|', or after FOR, LET or SOME

14.2. Scope of a Binding

We define the scope of a name as that part of a module specification where the name may be used in an expression, independently of the constraints imposed by type compatibility.

Generally, a binding is valid from the place where it was established up to some precise place, given below (although in some cases we give the scope itself rather than the place where the binding is destroyed).

Object	Binding
Global definitions	The whole module
Formal and result arguments	Valid only in the specification of the corresponding function or in the expression defining the macro
Locally bound variable	The expression(s) following the '¦', INSET or ':' after FORALL, EXISTS, LET, SOME, and in set and vector constructors.

An important additional comment is that, although it is permitted to alter a valid binding temporarily (e.g., by using the name of a formal argument as a locally bound variable), this will mask the original binding for the scope of the new one. It seems that this practice should be avoided, inasmuch as it can bring only confusion in the specification.

15. COERCION RULES

As has been said in Section 5, a united type is very close to being a type by itself, and normally an expression of a united type should not match an expression of one of the components of the united type. However, such compatibility is permitted in some cases. The determination of these cases is based on what is known as the "coercion rules".

The first case of coercion is the TYPECASE expression, which should be the normal (and only) way to coerce "downwards", i.e., from a united type to a component type. Upward coercion (i.e., when an expression of a simple type is implicitly converted to a united type containing the original type as one of its components) may take place in the following cases:

--When a formal argument to a function is of a united type, the corresponding actual argument may be of a component type.

--In an expression of the form a = b, where a is a quoted V-function reference, the result argument of the V-function being of a united type, and b is an expression of a component type.

The actual rule is the same as above for TYPECASE expressions and actual arguments to functions. It is more general in the other cases, where an attempt to match a united type against one of its components, or two distinct united types with some common components, is tolerated, albeit with a warning message which will hopefully remind the user that he/she is walking on dangerous ground. A warning message will also be issued when, in the constructs IF, SET, and

VECTOR, the types of the constituent expressions do not match. However, this will not be the case for the binary operators on sets, UNION, INTER, DIFF, INSET, and SUBSET. Thus, the general policy is to tolerate any expression that might be meaningful for some values of the variables or functions (as opposed to strong type-checking where an expression must be meaningful for all possible values).

16. COMMENTS

Although it has been barely mentioned so far, the language has a comment feature: A comment consists of the sign '$' followed by a sequence of characters without any space, parentheses, or square bracket; or by a string of characters enclosed within two double-quote signs (") (the string itself containing no double-quote); or by any sequence of characters enclosed within two matching parentheses or square brackets (in the case of parentheses, the sequence must not contain any right square bracket that is not preceded in the sequence by a left one). More briefly, a comment can be described as a '$' followed by any arbitrary LISP S-expression--see (Tei 75).

A comment may appear at any place where a space is legal (i.e., anywhere except in the middle of a syntactic unit).

17. MAPPING FUNCTION EXPRESSIONS

The specifications for mapping function expressions follow a slightly different syntax, although the "expression" part of the language is exactly the same.

17.1. Module Names

A list of mapping functions specifications should begin with

MAP module_1,..., module_i TO module_j,..., module_k;

where module_1 is the module containing the functions and parameters to be expressed as functions of elements of module_j,...,module_k; module_1,..., module_i are referred to as the "target" modules.

17.2. TYPES

The TYPES paragraph is similar to the one described for module specifications, and it abides by the same rules.

17.3. DECLARATIONS

The same conventions for deferred binding can optionally be applied and the DECLARATIONS paragraph provides the same facility towards this end.

17.4. DEFINITIONS

The same macro capabilities are provided in the DEFINITIONS paragraph.

17.5. EXTERNALREFS

An EXTERNALREFS paragraph similar to the one optionally used in a module specification is mandatory for mapping functions; it must contain declarations for all the primitive objects (i.e., designator and scalar types, parameters, and V-functions that are not derived) appearing in the mapping function.

17.6. INVARIANTS

The INVARIANTS paragraph is the counterpart of the ASSERTIONS paragraph for module specifications. Invariants are contained in a single paragraph in the mapping function specifications, and express constraints among the values of V-functions at the lower level. The syntax is similar to that of the ASSERTIONS paragraph of a module specification, save for the keyword INVARIANTS which is substituted for ASSERTIONS. Invariants must be proved as part of the output assertions to all implementing programs, but they can be used as input assertions to these programs. Verification of implementation correctness cannot be successfully performed unless strong input assertions (such as provided by the invariants) can be assumed.

17.7. MAPPINGS

The MAPPINGS paragraph begins with MAPPINGS. It contains a list of pairs of the form

<object> : <expression>;
or
<symbol> : <type specification>;

where <expression> is any expression satisfying the rules described earlier, and <object> is one of the following:

--The name of a parameter of one of the target modules

--The name of a scalar type, in which case <expression> must be a set expression consisting of a list of as many expressions as there are constants in the scalar type definition;

--A construct of the form

<symbol> (<argument list>)

where <symbol> is the name of a visible V-function or a parameter (with arguments) of the target module, and <argument list> is a list of formal arguments in the usual sense.

The same scope rules apply, except that there is no result argument.

The type of each expression, derived from the types of its various components declared either in the declarations or in the source modules, should be the same as that of the associated parameter or V-function in the target modules, or a set expression in the case of a scalar type name.

In the second form, the symbol should be the name of a designator type of one of the target modules, and the type specification should be legal in at least one of the lower modules.

18. THE TENEX-INTERLISP IMPLEMENTATION

The properties of specifications can be checked automatically by a "specification handler". Such a program has been written to run under the TENEX operating system on a PDP-10 machine; some of the conventions used (e.g., for strings) are a consequence of the conventions used in the implementation language (namely INTERLISP).

18.1. Strings

A string is the equivalent of an INTERLISP string, i.e., an arbitrary sequence of printable ASCII characters enclosed within two " (double-quote), where the characters " and % have to be represented as %" and %%.

18.2. Identifiers

An identifier (generally represented as <symbol> in this manual) is any sequence of up to 126 printable ASCII characters which is not a number, which does not contain any of the characters (,), [,], {, }, ¦, ,, ;, :, =, ~, -, +, <, >, *, /, ', `, ?, $, space; and which does not start with either " or #. Note that 1%&@! is a perfectly legal identifier.

18.3. File Inclusion

If the character # appears after any of the characters listed in the previous paragraph, the next expression (in the INTERLISP sense) will be interpreted as a file name, and the program will take its input from that file until an end of file is reached, at which point it will come back to the original file. This may be repeated, i.e., the included file may itself include

a third one, etc., although a file may not include itself, directly or indirectly.

19. CONCLUDING REMARKS

The reader may have noticed (by the very absence of such expressions) that nowhere have we spoken of "the value of a variable" or "the contents of a variable". Such phrases have been (carefully) avoided to stress one of the particularities of any specification language, i.e. the property of being nonprocedural; a name refers to a mathematical object, and never to an address or any other component of a computer. Indeed the language does not provide the concept of a pointer, and this could be introduced only as a designator.

SPECIAL is often modified to deal with previously unsuspected problems. Part of the work to be done in order to design a good specification language is to understand fully the concepts involved and the specification methodology on which the language is based. A very desirable goal would also be the complete axiomatization of the language constructs, similar to the work done on Pascal by Hoare and Wirth (Hoa 73). We hope to come up with such a formalization in the (more or less) near future.

ACKNOWLEDGEMENTS

The authors are grateful to the various members of the Computer Science Group at SRI for several suggestions. Of particular importance were the contributions of Dr. Robert S. Boyer. An important part of the earlier version of SPECIAL was the work of Bernard Mont-Reynaud.

The work described here has been supported by the US Air Force under Honeywell Purchase Order #9984718, by the Naval Electronics Laboratory Center under Contract N00123-76-C-0195, and by the National Science Foundation under Grant DCR74-18661.

- REFERENCES -

(Hoa 73) Hoare,C.A.R., and Wirth,N.; "An Axiomatic Definition of the Programming Language Pascal," Acta Informatica, vol.2, (1973).

(Neu 74) Neumann,P.G. et al.; "On the Design of a Provably Secure Operating System," Proc. Workshop on Protection in Operating Systems, IRIA, Rocquencourt, (August 1974).

(Neu 75) Neumann,P.G., et al.; "A Provably Secure Operating System," Final Report, Project 2581, Stanford Research Institute, Menlo Park, California, (June 1975).

(Par 72 a) Parnas,D.L.; "A Technique for Software Module Specification with Examples," Comm. ACM, vol.15 (May 1972).

(Par 72 b) Parnas,D.L.; "On the Criteria To Be Used in Decomposing Systems into Modules," Comm. ACM, vol.15 (December 1972).

(Rob 75 a) Robinson,L. et al.; "On Attaining Reliable Software for a Secure Operating System," Proc. 1975 International Conference on Reliable Software, Los Angeles, (April 1975).

(Rob 75 b) Robinson,L., and Levitt,K.N.; "Proof Techniques for Hierarchically Structured Programs," Stanford Research Institute (to be published).

(Tei 75) Teitelman,W.; "INTERLISP Reference Manual," XEROX Palo Alto Research Center, (December 1975).

APPENDIX A : GRAMMAR

Legend: The grammar for the specification language is given in the so-called extended BNF, where <...> means that the enclosed symbol is a nonterminal of the grammar; [...] means that the enclosed construct is optional; {...}* means that the enclosed construct can occur 0 or more times; {...}+ means 1 or more times, and {... | ... | ...} means an alternation, i.e., any of the constructs listed can be used.

For convenience, all the nonalphabetic terminals have been enclosed within simple quotes (').

<symbol> stands for any identifier, <number> for any integer or real number.

```
ROOT                    ::= MODULE <symbol> [<types>] [<declarations>]
                                [<parameters>] [<definitions>]
                                [<externalrefs>] [<assertions>]
                                [<functions>] END_MODULE
                        ::= MAP <symbol> { ',' <symbol> }*
                                TO <symbol> { ',' <symbol> }* ';'
                                [<types>] [<declarations>] [<parameters>]
                                [<definitions>] [<externalrefs>]
                                [<invariants>] [<mappings>] END_MAP

<types>                 ::= TYPES { <typedeclaration> ';' }+

<typedeclaration>       ::= <symbol> { ',' <symbol> }* ':'
                                { DESIGNATOR ¦ <typespecification>
                                  ¦ <setexpression> }

<typespecification>     ::= <symbol>
                        ::= INTEGER
                        ::= BOOLEAN
                        ::= REAL
                        ::= CHAR
                        ::= STRUCT '(' <declarationlist> ')'
                        ::= ONE_OF '(' <typespecification>
                                { ',' <typespecification> }+ ')'
                        ::= { SET_OF ¦ VECTOR_OF } <typespecification>

<simple declaration>    ::= <typespecification> <symbol>

<declaration>           ::= <simple declaration> { ',' <symbol> }*
                        ::= <symbol>

<declarations>          ::= DECLARATIONS { <declaration> ';' }+

<parameters>            ::= PARAMETERS { <parameterdeclaration> ';' }+

<parameterdeclaration>  ::= <typespecification> <symbol> [<formalargs>]
                                { ',' <symbol> [<formalargs>] }

<formalargs>            ::= '(' [<declarationlist>] ')'
                                [ '[' <declarationlist> ']' ]

<declarationlist>       ::= <declaration> { ';' <declaration> }*

<definitions>           ::= DEFINITIONS { <definition> ';' }+
```

```
<definition>          ::= <typespecification> <symbol> [<formalargs>]
                              IS <expression>

<externalrefs>        ::= EXTERNALREFS { externalgroup }+

<externalgroup>       ::= FROM <symbol> ':' { <externalref> ';' }+

<externalref>         ::= <parameterdeclaration>
                      ::= <symbol> { ',' <symbol> }* ':' DESIGNATOR
                      ::= <symbol> ':' <setexpression>
                      ::= { VFUN | OVFUN } <symbol> <formalargs>
                              '->' <declaration>
                      ::= OFUN <symbol> <formalargs>

<assertions>          ::= ASSERTIONS { <expression> ';' }+

<invariants>          ::= INVARIANTS { <expression> ';' }+

<functions>           ::= FUNCTIONS { <functionspec> }+

<functionspec>        ::= VFUN <symbol> <formalargs>
                             '->' <declaration> ';'
                             [<definitions>]
                             [{ HIDDEN ';' | <exceptions> }]
                             [<assertions>]
                             { INITIALLY | DERIVATION } <expression> ';'
                      ::= OVFUN <symbol> <formalargs>
                             '->' <declaration> ';'
                             [<definitions>] [<exceptions>] { <delay> }*
                             [<assertions>] [<effects>]
                      ::= OFUN <symbol> <formalargs> ';' [<definitions>]
                              [<exceptions>] { <delay> }*
                              [<assertions>] [<effects>]


<delay>               ::= DELAY [WITH { <expression> ';' }+]
                              UNTIL <expression> ';'

<exceptions>          ::= EXCEPTIONS { <expression> ';'
                             | EXCEPTIONS_OF <call> ';'
                             | RESOURCE_ERROR ';' }+

<effects>             ::= EFFECTS { <expression> ';' }+

<mappings>            ::= MAPPINGS { <mapping> ';' }+

<mapping>             ::= <symbol> [<formalargs>] ':' <expression>
```

```
                        ::= <symbol> ':' <typespecification>

<expression>            ::= IF <expression> THEN <expression>
                                ELSE <expression>
                        ::= LET <qualification> { ';' <qualification> }*
                                IN <expression>
                        ::= SOME <qualification>
                        ::= { FORALL | EXISTS } <qualif\declarationlist>
                                ':' <expression>
                        ::= TYPECASE <symbol> OF { <case> ';' }+ END
                        ::= <expression> <binaryoperator> <expression>
                        ::= { NOT | '~' } <expression>
                        ::= '-' <expression>
                        ::= '(' <expression> ')'
                        ::= <symbol>
                        ::= <number>
                        ::= <character constant>
                        ::= <string constant>
                        ::= TRUE | FALSE | UNDEFINED | ?
                        ::= <expression> '[' <expression> ']'
                        ::= <expression> '.' <symbol>
                        ::= { CARDINALITY | LENGTH | MAX | MIN | SUM
                                | INTPART | FRACTPART }
                                '(' <expression> ')'
                        ::= LOG '(' <expression> ',' <expression> ')'
                        ::= NEW '(' <symbol> ')'
                        ::= [EFFECTS_OF] <call>
                        ::= <structureconstructor>
                        ::= <vectorconstructor>
                        ::= <setexpression>

<qualif\declarationlist>::= {<qualification> | <declaration>}
                                { ';' <qualification> | <declaration> }*

<qualification>         ::= [<typespecification>] <symbol>
                                { '=' | '~=' | '<' | '<=' | '>=' | '>' |
                                  SUBSET | INSET | '|' } <expression>

<case>                  ::= <typespecification> ':' <expression>

<binaryoperator>        ::= '^' | '*' | '/' | INTER | '+' | '-' | UNION |
                                DIFF | '=' | '~=' | '>' | '>=' | '<' |
                                '<=' | INSET | AND | OR | SUBSET | MOD | '=>'

<call>                  ::= ['''] <symbol> '(' [ <expression>
                                { ',' <expression> }*] ')'

<structureconstructor>  ::= '<' [ <expression> { ',' <expression> }* ] '>'
                        ::= '<' <range> ':' <expression> '>'
```

```
<vectorconstructor>     ::= VECTOR '(' [ <expression>
                               { ',' <expression> }* ] ')'
                        ::= VECTOR '(' <range> ':' <expression> ')'

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

<setexpression>         ::= '{' [ <expression> { ',' <expression> }* ] '}'
                        ::= '{' [<typespecification>] <symbol> '|'
                               <expression> '}'
                        ::= '{' <expression> '..' <expression> '}'
```

APPENDIX B : RESERVED WORDS

The following list contains all the reserved words of the language, i.e., the identifiers that may NOT be used to designate arbitrary objects. The identifiers T and NIL, although not reserved, are not allowed in the current implementations.

AND
ASSERTIONS
BOOLEAN
CARDINALITY
CHAR
DECLARATIONS
DEFINITIONS
DELAY
DERIVATION
DESIGNATOR
DIFF
EFFECTS
EFFECTS_OF
ELSE
END
END_MAP
END_MODULE
EXCEPTIONS
EXCEPTIONS_OF
EXISTS
EXTERNALREFS
FALSE
FOR
FORALL
FRACTPART
FROM
FUNCTIONS
HIDDEN
IF
IN
INITIALLY
INSET
INTEGER
INTER
INTPART
INVARIANTS
IS
LENGTH
LET
LOG
MAP
MAPPINGS
MAX
MIN
MOD
MODULE
NEW
NOT
OF
OFUN
ONE_OF
OR
OVFUN
PARAMETERS
REAL
RESOURCE_ERROR
SET_OF
SOME
STRUCT
SUBSET
SUM
THEN
TO
TRUE
TYPECASE
TYPES
UNDEFINED
UNION
UNTIL
VECTOR
VECTOR_OF
VFUN
WITH

STANFORD RESEARCH INSTITUTE
Menlo Park, California 94025 · U.S.A.

THE HIERARCHY SPECIFICATION ENVIRONMENT

by

Robert Boyer

and

Olivier Roubine

TABLE OF CONTENTS

1. Introduction

This document presents the automated environment for manipulating specifications of system hierarchies developed to support the SRI methodology. The reader is supposed to be familiar with this methodology which is described in (Robinson et al.,75) and (Robinson and Levitt,75) . The aspects of the methodology which are most relevant to this document are summarized in section 2.

The environment described in the following sections was implemented in INTERLISP on the TENEX operating system. Although little needs be known about INTERLISP, some familiarity with the TENEX file system is desirable since the environment is fairly dependent on it. In particular, we use the following terminology:

in a file name

<DIR>FOO.EXT;n

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

It should be noted that the present system is a prototype. Any question, complaint or suggestion should be addressed to Olivier Roubine, Bernard Mont-Reynaud, or Robert Boyer, of the Computer Science Group, SRI.

2. The Methodology: A Brief Overview

This section is mainly intended to define some of the terms that will be used in the following sections, rather than to be a self-contained description of the methodology.

A large software system is decomposed into a number of small "modules". A module is likely to deal with one single problem, e.g., providing a certain data structure and operations for manipulating it. More precisely, a module is made of a list of "V-functions", that are functions returning a value but leaving the state of the world unchanged, and a list of "O-functions" which perform operations on the state of the world. A module may also have "parameters"

which are particular components of the state of the world which may have different values in different worlds, but the value of which cannot be changed by any O-function (in fact, parameters can be viewed as values returned by V-functions, which can never be changed.) A module may also deal with a concept referred to as a "designator" which introduces a new type of object (for more details, see (SPECIAL Reference Manual)).

V-functions can be either visible or hidden, and either primitive or derived. We sometimes refer to the objects of a module as the set of V- and O-functions, parameters, and designators of the module. The primitive objects are the parameters, designators, and the primitive (non-derived) V-functions.

Modules can be "implemented" in terms of objects of other modules. O-functions are implemented by an "abstract program", an aspect of the methodology which is not dealt with in the present environment. The other objects are implemented via mapping-function expressions. Leaving aside the problems associated with abstract programs, we shall (somewhat improperly) say that a module M0 is implemented on M1, M2,..., Mn, if for each primitive object of M0, there is a mapping-function expression which uses only objects of M1, M2,..,Mn.

How all the modules are grouped and "implemented" is what will define a "hierarchy". Our environment does not deal however with hierarchies of modules. Rather, we introduce an intermediate concept: the notion of an "interface". A module specification generally corresponds to a conceptual abstraction. An interface could be associated with a functional abstraction: it corresponds to a set of modules which is self-contained with respect to external references and which is sufficient to implement other interfaces in a particular hierarchy.

An interface could be viewed as a set of objects (the same kind of objects as those defined by modules). We prefer to see it as a set of modules, all the objects of which are available except for some of them which are explicitly hidden: an interface is something like a "pool" of objects provided by a set of contributors: the modules of the interface. Two criteria must be satisfied.

- Internal completeness: if a module of the interface has external references, those must be to other modules of the interface.

- External completeness: all the modules of an interface must be implemented in terms of modules of one single interface.

There is a naming problem within an interface since the same interface might contain several instances of the same module specification. Although this problem can be solved by adequate tools in the specification language, the present environment does not cope with it and requires as many module specifications as module

instances, and the names of all the objects of an interface must be unique.

An interface I0 "implements" an interface I1 with a set of mapping-functions F1, F2...Fk if for a module M1 of I1, there is a mapping-function from some modules of I0 to M1.

A "Hierarchy" can now be defined as a set of interfaces such that, with the exception of a particular one, all the interfaces are implemented by some other interface of the hierarchy, with some set of mapping-functions (to be provided). Note that because of the external completeness criterion, a hierachy has necessarily the structure of a rooted tree where nodes are interfaces and edges correspond to mapping-functions.

3. General Presentation of the System

From the above description, it is clear that in order to define a hierarchy, one must introduce specifications for modules, mapping-functions, interfaces and the hierarchy tree. Generally, each of these entities must satisfy some criteria for internal consistency, and additional criteria for external consistency, i.e., consistency with respect to other components of the hierarchy.

Our system consists of various functional units which perform the necessary consistency checks on a hierarchy.

The system is invoked by typing:

<HIER>NS.;
 followed by a carriage return

to the TENEX EXEC. This will invoke a single LISP subsystem which contains commands for each functional unit of the environment.

The functional units are the module-handler, the mapping-function handler, the interface-handler and the hierarchy-handler. Each unit takes as input one or more TENEX files and produces either error diagnostics corresponding to the failure of some consistency rule, or a file, called a "link file", which records the successful completion of the particular set of consistency checks performed by this functional unit. The exact contents of these link files needs not be described in this document. They contain some information (e.g., symbol tables) pertaining to some part of the system, and explicit references to the files from which this information was

gathered. These references are used to verify the consistency of a set of files.

As a rule, a link file is produced only when all the relevant checks have been successfully carried out: its existence in a directory is a "prima facie" evidence of the correctness of some part of the system. However, it might very well be the case that after having checked, let's say, an interface, the user changes the specification of some module without checking the interface again. 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. Such a situation will be detected automatically; it can then be remedied by performing all the necessary checks to create up-to-date link files .

4. The Module Handler

The module handler is the set of functions that has sometimes been described as the "specification handler", although this term is equally applicable to the handling of Mapping Function specifications.

4.1. Source File

The module handler expects a source file containing a module specification written in the specification language SPECIAL. The source file must be named

modulename.SOURCE (1)

(Note that SPECIAL does not accept hyphens in a module name, whereas TENEX does not accept underbars in a file name. As a consequence, if the name of the module contains underbars, these should be translated to hyphens in the corresponding file name.

4.2. Commands

4.2.1. Consistency Check:

CHECKMODULE(filename)

is the function that performs all the checks on the module specification, to verify that it complies with the rules of the

---

1) Throughout this document, any term ending with the suffix "name" refers to a symbolic name of some sort.

specification language, described in (SPECIAL Reference Manual). This function operates in two phases:

--A syntactic analysis of the contents of the file given as argument is first attempted; if a specification is syntactically incorrect, a message of the following form will be printed:

```
**************************************************************************
Illegal lexeme  level (of type SYMBOL) in
 ...me, level, cat_set)) + i-j+1
      > max_message !!!level!!!; cat_set);
BOOLEAN no_message (up; level; cat_set

A lexeme of one of the following types was expected:
TO  ;  )  ADDOP  -  ,  OR  AND  :  =>  MULTOP  [  ]  THEN  ELSE  IN
INSET  RELOP  <  >  RIGHTBRACE  .  (
What should be done?
**************************************************************************
```

The syntactic token which is not accepted by the analyzer appears between !!!....!!! in a "window" of 100 characters of the source text. At this point, the user may choose to replace the offending lexeme by an arbitrary string of his choice, by answering R followed by the new text appearing between two " characters. This will cause not only the error to be corrected so as to be able to continue processing the text, but will also generate a new version of the source file, in which the error will have been corrected.

Another alternative is for the user to type T (for 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; exiting the editor (by "ex" or ";h") will return control to CHECKMODULE: at this point, the processing can be resumed either where it has been interrupted (i.e., before the offending lexeme), by typing D (for Default), or at the beginning of the file, by typing S (for Start over), or at an arbitrary location, specified by a number interpreted as a byte address in the file. Note that this latter approach is very unreliable, unless the user is perfectly sure of what he is doing, because the internal state of the system will not be cleaned; in particular, it is not possible to "backtrack", i.e., to forget some of the lexemes that have already been processed.

The last alternative is A (for Abort) which will interrupt the function and go in error mode.

When the whole file has been successfully processed by the syntactic analyzer, the full file name is printed, and the second phase is initiated.

--The second phase performs all the so-called semantic checks, and

is mostly concerned with the enforcement of the type rules, the scope rules, and the well-formedness of the specification. Various error or warning messages may be issued, giving some information about the location of the error (e.g., a definition or a function specification), the kind of error (e.g., type incompatibility, scope error, etc...), and sometimes a context information which is an expression derived from the original one --generally a prefix form of the source expression, e.g.,

a + b
would appear as
(+ a b)

If no error was detected during the second phase (warning messages are not considered as resulting from errors), the following message is printed:

Link File?

Answering Y will cause a link file modulename.MLINK to be written, and its name to be returned as the value of CHECKMODULE; if the answer is N, the value is T.

In the case of external references, however, the check will be final only after the information contained in the EXTERNALREFS paragraph has been matched against the original definition. This is not done by CHECKMODULE, but by the interface-handler.

4.2.2. Printing a module specification:

REFORMAT(inputfilename outputfilename)

is a pretty-print routine which causes a module specification contained in the file "inputfilename" to be printed on the file "outputfilename" with appropriate spacing and indentation. If the arguments are omitted, the program will ask explicitly for them to be provided. A possible use of this function is to give it the value T as its second argument, which will cause the specification to be printed on the user's terminal; it should be noted, however, that the command

TTY filename

can be typed directly to the hierarchy manager, and will cause an arbitrary file "filename" to be printed on the terminal(2).

---

2) This facility, as well as the direct invocation of TECO by typing "TECO filename" are part of the standard initialization of the system; if the user types GREET(), as suggested by the system message, he/she is likely to lose these two commands.

5. The Mapping-Function Handler

The mapping-function handler is separated in two functional units: one deals with the internal consistency of a mapping-function specification, and the other, which is similar to a second pass, checks the consistency of the mapping function expressions with respect to the modules involved.

5.1. Internal Consistency Checker

The internal consistency checker is the counterpart of the module specification handler for mapping-functions: the same commands for input and output are available.

5.1.1. Source file

The source file is a mapping-function specification which associates an expression using objects of "lower" modules with each primitive object of an "upper" module. The detailed syntax is described in (SPECIAL Reference Manual).

The name of such a file should be

filename.MAP

5.1.2. Check

The source file can be read in by the command PARS, as in the case of a module specification. The internal form will also be left in the variable MODULE, and then the checks are invoked by the command

CHECKMAPSPEC(filename)

which runs through the same three passes as CHECKMODULE does. At the end of the checks, if the user wishes to create a link file, a filename will be asked for, and the system will create a new source file filename.MAP, and a link file filename.SYNLINK.

5.2. External Consistency Checker

The external consistency checker is invoked by the command

CHECKMAP(filename)

and will attempt to take its input from the link file filename.SYNLINK.

The checks are:

--file system consistency: each of the modules referenced in the mapping function should have been previously checked (i.e., there exist an "MLINK" file), and the corresponding source files should not have been modified since then;

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

--completeness: there must exist a unique mapping-function expression for each primitive object of the "upper" module. This check can fail in three ways: an object may have no representation, or there may be an object appearing in the mapping-function specification and 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 automatically created under the name

filename.SEMLINK

6. The Interface Handler

In Section 2, we have given a definition of the term "interface" and have introduced the rules to be satisfied. The interface handler enforces these rules.

6.1. Source File

An interface specification is to be written on a file according to a particular syntax:

```
<interface specification> ::= (INTERFACE <interfacename>
                                  <module list>)
<module list> ::= <module dec>
              ::= <module list> <module dec>
<module dec>   ::= <modulename>
               ::= (<modulename>)
               ::= (<modulename> WITHOUT <objectlist>)
<objectlist> ::= <objectname>
             ::= <objectlist> <objectname>
```

When a module name is followed by the construct "WITHOUT <objectlist>", each name apearing in the list is taken to be hidden by the interface. If this construct is not present, all the global objects of the module are visible at the interface.

The interface specification is expected to be found in a file named

interfacename.INTERFACE

6.2. The Interface Checker

The command

CHECKINTERFACE(interfacenam^)

will cause the file interfacename.INTERFACE to be read and the following checks to be performed:

--A syntax check controls the well formedness of the specification (it also checks that the prefix of the filename corresponds to the interface name).

--The consistency of the file system is also checked, i.e., all the modules apearing in the interface specification must have been specified, checked, and should not have been modified after the check.

--The naming consistency is enforced: presently, the names of all the objects of the modules of an interface must be unique.

--The objects appearing on a "WITHOUT" list must have been defined in the specification of the corresponding module.

--Lastly, the program verifies that the interface is "closed" under external references, i.e., that all the externalrefs of any module are to objects of other modules of the interface and that the types correspond.

If all these checks are carried out correctly, a link file

interfacename.ILINK

is automatically 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 be considered incomplete, and no link file will be produced.

7. The Hierarchy Handler

A complete software system is described in our methodology as a hierachy of interfaces (see Section 2). Within a hierarchy, two interfaces are connected if there is a set of mapping-functions which implements all the modules of one interface in terms of objects that are visible in the other one.

A hierarchy is specified when all the interfaces and all the mapping-functions are known. This information is to be found on a file named

hierarchyname.HIERARCHY

7.1. Source File

The syntax for a hierachy specification is given formally below:

```
<hierarchy specification>::= (HIERARCHY <hierachyname>
                                  <list of implementations>)

<list of implementations> ::= <implementation>
                          ::= <list of implementations>
                                  <implementation>

<implememtation> ::= ( <interfacename> IMPLEMENTS <interfacename>
                           USING <list of mapping-functions>)

<list of mapping-functions> ::= <mapping-function name>
                            ::= <list of mapping-functions>
                                  <mapping-function name>
```

Some additional restrictions must be observed:

--<hierachy name> should be the same identifier as the prefix of the file name.

--<mapping-function name> is the prefix name of a file containing a mapping-function specification .

--The order in which the implementation specifications are given is not arbitrary: in a specification

(IO IMPLEMENTS I1 USING ---)

we shall refer to IO as the lower interface and I1 as the higher one. The lower interface of the first implementation

specification must be the root of the hierarchy, and any lower interface in a specification must either be the root or have appeared as the higher interface of a previous specification. In addition, each interface except the root must appear once and only once as the higher interface of some implementation specification (these restrictions guarantee that the hierarchy is an oriented tree with the lowest interface as its root.

7.2. The Hierarchy Checker

The Hierarchy checker is invoked by the command

CHECKHIERARCHY(hierarchyname)

It will first check that the hierarchy specification found (hopefully!) in the file hierarchyname.HIERARCHY is syntactically correct and that none of the rules described in 7.1. is violated.

It then verifies that for each implementation specification, link files exist for both interfaces and all mapping-functions and that the implementation is consistent and complete.

The implementation is consistent when the higher module of each mapping-function exists in the higher interface and the lower modules exist in the lower interface. The implementation is complete if there is one mapping-function for each module of the higher interface. If this is not the case, but if all the modules of the higher interface which have no corresonding mapping-function also appear in the lower interface the program will assume that the mapping is the identity mapping. Thus, modules which are mapped identically need not be associated with a specific mapping-function file.

Note that a module may appear in several interfaces, and that a mapping-function specification may be used in several implementations.

If the checks have been carried out properly, you are seeing the light at the end of the tunnel: the letter T will be printed and a file

hierarchyname.HLINK

will 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 a file system.

- REFERENCES -

(Robinson et al.,75): L. Robinson, K.N. Levitt, P.G. Neumann and A.K. Saxena; "On attaining reliable software for a secure Operating System" - Proc. 1975 International Conference on Reliable Software, Los Angeles, April 1975

(Robinson and Levitt, 75): L. Robinson and K.N. Levitt; "Proof Techniques for hierarchically structured Programs" SRI,1975 (to appear in Comm. ACM)

(SPECIAL Reference Manual): O. Roubine, B. Mont-Reynaud and L. Robinson; "SPECIAL Reference Manual" SRI Memo, February 1976

DISTRIBUTION LIST

Mr. Tony Allos Code 6201 Naval Electronics Laboratory Center 271 Catalina Boulevard San Diego, CA 92151	1 copy
Mr. L. Sutton Code 5200 Naval Electronics Laboratory Center 271 Catalina Boulevard San Diego, CA 92151	35 copies
Mr. William Carlson Advanced Research Projects Agency Office of Secretary of Defense 1400 Wilson Boulevard Arlington, VA 22209	15 copies
Mr. Neal Hampton Code 5200 Naval Electronics Laboratory Center 271 Catalina Boulevard San Diego, CA 92151	15 copies
Professor Stuart Madnick MIT - Sloan School E53-330 Cambridge, Mass 02139	1 copy
Mr. John Machado The Naval Electronic Systems Command National Center No. 1 Crystal City, Washington, D.C. 20360	5 copies


FILM
51