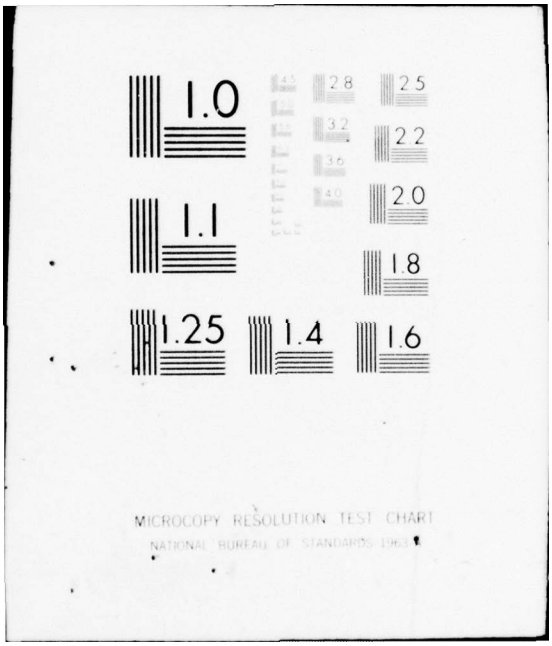
AD-A038 255 STANFORD RESEARCH INST MENLO PARK CALIF COMPUTER SCI--ETC F/G 9/2
SPECIAL - A SPECIFICATION AND ASSERTION LANGUAGE.(U)
SEP 76 L ROBINSON, O ROUBINE N00123-76-C-0195
UNCLASSIFIED CSL-46 NL

1 OF 1
AD A038255

END

DATE FILMED 5-77


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

Technical Report CSL-46

January 1977

SPECIAL - A SPECIFICATION AND ASSERTION LANGUAGE

By: LAWRENCE ROBINSON and OLIVIER ROUBINE

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 - A SPECIfication and Assertion Language

5. TYPE OF REPORT & PERIOD COVERED
Technical Report

6. PERFORMING ORG. REPORT NUMBER
CSL-46

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
September 1976

13. NO. OF PAGES
41 pages

15. SECURITY CLASS. (of this report)
Unclassified

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)
41p.

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)
Approved for public release; distribution unlimited.

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

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Formal specifications, Modularity, Hierarchical structure, Abstract machines, Logic, Language design

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)
SPECIAL is a specification language developed in conjunction with the SRI methodology for design, implementation, and formal verification of software systems. Some of the language features are specific to the SRI methodology. Others, such as its non-procedural nature, concept of type, and aggregate data types, are generally useful for software specification and verification. A description of the features of the language is supplied, along with several examples of its use. The language has proved useful in the design of several large software systems, including an operating system. A discussion of the issues in the design of SPECIAL is presented, followed by a description of its features and some examples.

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

I. INTRODUCTION

SPECIAL (SPECIfication and Assertion Language) is a language for specifying software systems and was created with the following goals in mind:

- To specify systems in conjunction with a particular methodology for design, implementation, and proof of computer systems. The methodology, called the SRI Hierarchical Design Methodology, is described elsewhere [6, 7].
- To be powerful enough to specify a large class of systems, including systems containing both hardware and software, yet restrictive enough to permit syntactic checks of well-formedness of a specification.
- To be usable directly in the statement and formal proof of properties of software systems.
- To foster abstraction through the hiding of unnecessary data and algorithms of implementation.
- To specify systems to be implemented in any programming language.

SPECIAL describes properties of the data contained in a software system, but not properties concerning the time required to perform an operation.

In order to fulfill these goals, we have developed a language with the following features:

- An inner level that allows the writing of non-procedural expressions, using predicate calculus and set theory. This level is called the assertion level of the language. Expressions at the assertion level are used to describe the behavior of software systems and their abstract properties in a precise way.
- An outer level, called the specification level, that enables the

description of a software system as a hierarchy of modules, in which each module is an abstract machine having a state and operations to change the state. The behavior of a module is described in terms of a formal specification, containing functions (2) that can be called by programs using the module. The state of a module is represented by the outputs of its V-functions (functions that return values). The state transformations of a module are called O-functions (functions that perform operations) of the module. Each transformation is described as a set of assertions relating the values of V-functions before the function call to the values of V-functions after the call. OV-functions are functions that both return values and transform the state. In addition to module specifications, assertions written in SPECIAL are used to specify relations among the states (V-function values) at different levels of the hierarchy. These relations are called mapping functions.

- Its own notion of type and facilities for describing abstract data types.
- Aggregate objects such as sets, vectors, and structures.
- An expression-based macro facility.
- Features to facilitate the characterization of objects without overconstraining them, and the detection of abnormal conditions.

Section II of this paper describes some background in the design of SPECIAL. Section III describes the assertion level. Section IV describes the specification level. Section V presents two complete examples of module specifications.

---

(2) The term "function" hereafter refers to a V-, O-, or OV-function of a module and not to a mathematical function.

II. BACKGROUND AND GENERAL CONCEPTS

A. Background

Formal specification languages of some kind have always been necessary for research in programming language semantics and program verification. Recently there has been some emphasis on specification languages themselves -- as design tools for software systems. Parnas [4, 5] first suggsted the idea of designing a software system as a collection of formally specified modules. Parnas' language was more precise than the informal methods of software specification that preceded it, but its syntax and semantics were not formally stated. Recently there have been several efforts [1, 2] aimed at specifying the formal properties of data structures and other software systems. Yet oher specification languages have emphasized the theory of primitive recursive functions [10] and the formal English of a mathematics textbook [11] for describing properties of software.

The SRI Hierarchical Design Methodology suggests a particular (hierarchical) way of structuring these modules in order to improve the reliability of large software systems by:

- Formally stating all design decisions.
- Allowing a complex design to be structured so that it can easily be understood.
- Allowing proofs of formally stated properties of the design.
- Allowing a proof of syntactic consistency of the implementation with the design specifications.

Several small proofs have been completed (e.g. [6, 9]), and proofs for the design and implementation of a general-purpose operating system whose major design goal is security [3, 7], are now in progress.

SPECIAL was developed through numerous attempts to write module specifications in the style of Parnas ([4]) for many different kinds of software systems. This paper presents a self-consistent and working version of the language, although it may change as more experience is gained.

In the description of SPECIAL that follows, some of the examples have been over-simplified for ease of explanation. One feature of SPECIAL that is not discussed deals with the handling of parallelism. However, the full grammar appears in the Appendix, and a complete description of the language appears in the SPECIAL Reference Manual [8].

B. General Concepts

The problem of designing a good specification language is immense. There are perhaps as many factors as there are in designing a good programming language, and there is not much experience of others to draw on. The language should be able to characterize the objects of the system it is specifying (e.g., the variables, procedures, and data structures), and may employ objects of its own to facilitate that description -- thus the distinction between specification objects and implementation objects. Specification objects are not computed, and may not be computable, in the implementation. The language must be well suited to the particular kinds of systems it hopes to specify (e.g., operating systems, data base systems), and at the same time must adhere to the constraints of the design methodology being used with the language. The language must allow as much machine checking of specifications as possible, and also be amenable to formal mathematical proof. The language must be powerful, but without a proliferation of specialized syntactic features to suit every possible need. The language must be conceptually

elegant, but must have features that make specifications in the language easy to read and write.

We shall now outline some of the desirable properties of a specification language, with the intent of later motivating each of the features of SPECIAL according to the list.

- Mathematical basis: A specification language must have some features that relate directly to the mathematics on which the language is based. In the case of SPECIAL, the mathematics involved is that of set theory and first-order logic (including quantification), with integer and real arithmetic. SPECIAL allows the writing of arbitrary expressions in these domains. Other specification languages, (e.g., [1] and [10]) may not permit such generality.
- Powerful, concise expressions: A specification language should be able to state, in a straightforward way, properties shared by different objects. Sets, vectors, and structures (as well as their constructors) are supported by SPECIAL in order to assist the writing of these expressions.
- Well-suited to the specified systems: A specification language should be able to accomodate in a natural way the properties of the systems being specified. For example, in operating systems, it is desirable to be able to look at a machine word as being of different types under different circumstances. The concept of type in SPECIAL provides this facility (united types) without restricting the kind of machine checking that can take place.
- Formal statements: It is desirable in a specification language to be able to make as many formal statements about an object as possible. In

SPECIAL one statement that can always be made about an object is its type (i.e., the set of values that it can assume). The macro facility in SPECIAL provides a formal substitution rule based on expressions. O-functions in SPECIAL are defined completely by the specification: every V-function value not mentioned as changing in the specification must stay the same.

- Characterizing an object: Although the above goal (formal statements) is sometimes desirable, some specification language objects must not be overconstrained for fear of dictating a particular implementation (e.g., a particular ordering among elements of a vector). Thus, a specification language should be able to introduce new objects without uniquely defining them. The existential quantifier in SPECIAL allows this characterization, and we have introduced two other constraints (LET and SOME) that make this process more straightforward and readable.
- Abstraction and protection: One original goal of SPECIAL was to foster abstraction, i.e, the definition, maintenance, and protection of objects of an abstract data type [2]. Some specification languages (e.g., [1]) deal explicitly with the objects themselves, and write axioms about functions defined on the objects in order to specify the abstraction. In SPECIAL, an object is defined by the V-function values that take the name for the abstract object as an argument. The operations on an abstract object are defined by the O-functions that take the name for the abstract object as an argument. SPECIAL provides a facility for defining protected names (called designators) for these abstract objects, such that the creation and modification of these names is limited.

- Support a methodology: It is extremely useful to couple a specification language with a methodology for designing, structuring, and implementing systems. Not only does it allow the language to help constrain the systems to conform to the methodology (a desirable trait), but it helps restrict the possible alternatives in specification language design, which are many. This has characterized the relationship of SPECIAL to the SRI Hierarchical Design Methodology.
- Support readability and writability: In certain cases there is no possible justification for the inclusion of a feature in a specification language other than that the feature makes the specifications either more readable or writable. These features come under the general category of shorthands and mnemonics. Excessive length has been a problem with this specification technique in the past, so shorthands, such as macros and global declarations, are welcome for that reason. These shorthands also make it possible for the user to establish mnemonic names for macros and variables that are used globally. Concerning readability, other specification methods may produce more concise specifications (e.g., [1]): but we believe that they are more difficult to understand than those written in SPECIAL, especially when trying to implement a complex system based on them or when trying to ascertain whether or not they conform to the intentions of the specification writer.
- Machine checking: Besides the usual grammatical checking, it is also desirable to be able to check properties of variables such as their types, declarations, and bindings. Thus, SPECIAL provides mechanisms for separating the declaration of a variable from its use, and scope

rules that limit naming ambiguities (see Section IV-B). Machine checking of these rules is easily done. In fact we have implemented several on-line tools at SRI to perform syntactic checking. The use of such tools has reduced the frequency of errors (including logical errors) in writing specifications, by immediately flagging syntactically meaningless statements.

- Verification: The specifications in SPECIAL have already been used in proofs of correctness (by hand) of small hierarchical systems [6,9]. A semi-automatic verification system based on SPECIAL as an assertion language is currently in the planning stages. All of the constructs at the assertion level of SPECIAL can be formally stated. The constructs at the specification level of SPECIAL are in the process of being defined, as part of a formal description of the SRI Hierarchical Design Methodology.

The next two sections should be read with the preceding criteria in mind.

III. THE ASSERTION LEVEL

A. Introduction

Assertions (or predicates) in SPECIAL are used to describe properties of systems, e.g., state transformations, error conditions, invariant properties, mapping functions, and conditions that must be true at a particular time in a program's execution. This section describes the objects of the assertion level, and then describes the various operators of the language.

B. Expressions

The primitive construct at the assertion level is an expression, defined

as either 1) a constant or a variable, or 2) an ordered pair consisting of an operator and a sequence of expressions denoting its operands. A constant represents a single value, while a variable may represent several different values. Operators may be predefined (e.g., + or NOT) or user-defined (e.g., functions or macros defined by the user). The constants, variables, and operators of SPECIAL are called the objects of the language.

C. Types

Every object and expression in SPECIAL has a type. A type can be thought of as a set of values (or constants in the assertion language). With the exception of UNDEFINED as explained below, the set of constants in SPECIAL is partitioned. The type of a constant is of course the partition to which it belongs. For a variable, its type is the set of values that it can assume. For any operator, its operands as well as its result have a type. Thus, the type of an expression is either 1) if the expression is defined by a single constant or variable, then the type of the constant or variable; or 2) if the expression is an operator and a sequence of operands, then the type of the result of its operator. The types of all objects are explicitly specified: the type of each constant and predefined operator is specified as part of the semantics of SPECIAL; the type of each variable and user-defined operator in a specification must be declared by the writer of the specification.

In SPECIAL, we have incorporated a more flexible attitude in restricting the types of the operands for the predefined operators, than has been done in most strongly typed programming languages (e.g., [12]). The only restrictions imposed are those that would prevent the writing of meaningless expressions. We have also supplied a mechanism in SPECIAL (the TYPECASE expression) for

going back and forth between related types, so the term "coercion" has no meaning here.

The definition of a valid type in SPECIAL is related to the ways of naming sets of values in the language. Types in SPECIAL are of three varieties: primitive types, subtypes, and constructed types. A primitive type is a set of values that is disjoint from every other primitive type (with the exception of UNDEFINED) and such that no value in the set is defined in terms of any other values. There are three kinds of primitive types in SPECIAL:

- Those types whose values have well-known mathematical properties, called predefined types. The predefined types in SPECIAL are BOOLEAN, INTEGER, REAL, and CHAR. The usual kinds of operations apply to these types.
- Those types whose values are used to name objects of an abstract data type in specifications for abstract machines. These types are called designator types, and the objects of such types are called designators. The only operations that apply to designators are equality, inequality, and NEW(t), which returns a never-used designator of type t.
- Those types whose values are a set of symbolic constants, called scalar types. For example, the scalar type "primary_color" could be the set {red, blue, yellow}. There is no added generality in including scalar types in SPECIAL, because the objects of a scalar type could always be represented by the integers. However, scalar types provide a useful mnemonic, and increase reliability by restricting the operations that can be performed on objects of scalar type to equality and inequality. Thus, for example, one could never have the expression "red + blue = yellow", which could occur in an integer representation.

The other types can be built using the primitive types as a basis. A subtype is an arbitrary subset of a given type. An example of a subtype is the even integers. A constructed type can be any one of the following:

- An aggregate type, i.e., a set or vector of objects of the same type.
- A united type, which is the union of several types.
- A structured type, which is the cartesian product of types. For example, a complex number can be thought of as a structure of "REAL X REAL".

Aggregate and structured types provide complex specification language objects, whose use often shortens the specifications. The types out of which a subtype or constructed type is made are called the constituent types of the subtype or constructed type. The operations on a subtype are the same as the operations on its constituent type. However, syntactic checking cannot be made to determine whether or not an expression of a given type is a member of a subtype of that type. The operations defined for constructed types will be discussed in later subsections.

In addition to the standard constants of all types, there is a constant, UNDEFINED (or ?), that is a member of any type, but different from any other constant of any type. The semantics of this constant are that an object whose value is UNDEFINED really has no value. For example, the value of a stack pointer of a stack that does not exist would be UNDEFINED.

D. Declarations

A variable or user-defined operation must be declared, or associated with a type, before using it. The syntax of a declaration, written in extended BNF

(3), is

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

A type specification can be either an symbol that refers to a type or an explicit type specification. Here are some examples of explicit type specifications:

predefined:	INTEGER BOOLEAN
scalar:	{red, green, blue}
subtype:	{INTEGER x | x MOD 2 = 0} (the set of all integers x such that x MOD 2 is 0, i.e., the set of all even integers)
aggregate:	SET_OF INTEGER VECTOR_OF BOOLEAN
structured:	STRUCT(REAL realpart, imagpart)
united:	ONE_OF(INTEGER, VECTOR_OF CHAR)

A named type is a type that has been associated with a symbol for use in later declarations, e.g.,

```
STRUCT(REAL realpart, imagpart): complex_number
```

Then a new complex number xx can be declared as follows:

```
complex_number xx
```

A designator type must be a named type, so we write

```
stack_name: DESIGNATOR
```

to define the type, and later

```
stack_name st
```

to declare a variable of type "stack_name".

---

(3) In extended BNF, <...> 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 alternative among the enclosed constructs. All special characters that are terminal symbols have been enclosed in single quotes (e.g., ':').

A function is declared with its function type, name, formal arguments, and result (in the case of a V-function). For example, a V-function declaration looks like this,

```
VFUN read(segment s, INTEGER i) -> machine_word w
```

where "segment" and "machine_word" are named types, s and i are the formal arguments, and w is the result. An O-function declaration looks like this,

```
OFUN write(segment s, INTEGER i, machine_word w)
```

The exact role of type specifications and declarations in a module specification is discussed in Section IV.

E. Simple Operations on Predefined Types

These include the logical, arithmetic, and relational operators, which apply to objects of types INTEGER, BOOLEAN, and REAL. The logical operators are AND, OR, NOT, and IMPLIES, all of which take BOOLEAN arguments and have BOOLEAN results. The basic arithmetic operators are +, - (unary or binary), *, and /. They operate on objects of types INTEGER or REAL, hereafter called numbers. Objects of both types can be arbitrarily intermixed as arguments to the arithmetic operators, with the following constraints on the results: a binary operator having two INTEGER or two REAL arguments has an INTEGER or REAL result, respectively; and a binary operator having mixed arguments has a REAL result. The relational operators are =, ~=, >, >=, <, and <=, take numbers as arguments, and have a BOOLEAN result. For objects of type CHAR there are no distinguished operators: perhaps at a later time it will be advisable to define, as part of SPECIAL, a lexical ordering function, or a function that maps from a character to its integer code.

F. Operations on Sets, Vectors, and Structures

SPECIAL provides the conventional set operators -- union, intersection, set difference, elementhood, a subset predicate, and the number of elements in a set -- as UNION, INTER, DIFF, INSET, SUBSET, and CARDINALITY, respectively. The arguments to UNION can be sets of any type, and the type of the result is the set of the union of the constituent types of the arguments. For example, if the types of s1 and s2 are "SET_OF INTEGER" and "ONE_OF(SET_OF BOOLEAN, SET_OF CHAR)", respectively, then the type of "s1 UNION s2" is "ONE_OF(SET_OF ONE_OF(INTEGER, BOOLEAN), SET_OF ONE_OF(INTEGER, CHAR))". The arguments to INTER must be sets whose constituent types are not disjoint, and the type of the result is the set of the intersection of the constituent types of the arguments. For example, if the types of s1 and s2 are "SET_OF ONE_OF(INTEGER, BOOLEAN)" and "ONE_OF(SET_OF INTEGER, SET_OF CHAR)", respectively, then the type of "s1 INTER s2" is "SET_OF INTEGER". The arguments to DIFF must both be sets whose constituent types are not disjoint, and the result has the type of the first argument. The second argument to INSET must be a set, the first argument must be of a type that is not disjoint with the constituent type of the second argument, and the result is BOOLEAN. The arguments to SUBSET must be sets whose constituent types are not disjoint, and the result is BOOLEAN. CARDINALITY allows any set as an argument and returns an INTEGER result.

Constructors are expressions that define sets, vectors, and structures in terms of objects of their constituent types. An extensional constructor (used for all of the above types) requires the individual elements. An intensional constructor (used for sets and vectors only) supplies a necessary and sufficient property of the elements. The syntax of the extensional constructor for sets is as follows,

{1, 3, 5, 7}

The syntax of the intensional constructor for sets looks like this,

```
{INTEGER i | 0<i AND i<9 AND  i MOD 2 = 1}
```

This reads, "the set of all integers i such that 0 is less than i and i is less than 9 and i MOD 2 is equal to 1," or "the set of all odd integers on the open interval (0, 9)." Only the intensional constructor can be used to specify infinite sets, e.g.,

```
{INTEGER i | i MOD 2 = 1}
```

There are only two operations on vectors: length and extraction. The length operation is written LENGTH(v), where v is a vector expression. The result is of type INTEGER. The extraction operation is written v[i], where v is an expression of type "VECTOR_OF x" and i is an expression of type INTEGER, whose value is between 1 and LENGTH(v). The result is of type x. The extensional vector constructor is written as follows,

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

The intensional constructor for the same vector as above is written

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

Structures have an extractor. If a structured type has the following declaration,

```
STRUCT(REAL realpart, imagpart): complex_number,
```

then the two extractors for the type are "x.realpart" and "x.imagpart", where x is an object of type "complex_number." The extensional constructor for an object of type "complex_number" whose realpart is 1 and whose imagpart is 2 would be "<1,2>".

G. Quantified Expressions and Characterization Expressions

The syntax of a quantified expression is as follows:

{FORALL|EXISTS} <qualification> {':' <qualification>}* ':'
<expression>

where

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

The purpose of a qualification is to optionally restrict the domain of a quantified expression. This adds no generality but improves readability. For example, we write

FORALL x INSET s : p(x)

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

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

We write

FORALL x | q(x) : p(x)

to mean

FORALL x : q(x) => p(x)

Sometimes it is desirable to write expressions in which an object with a particular value or property is used repeatedly. To do this we have devised a construct called a characterization expression, in which a variable is first characterized and then used in an expression containing it. One of these, the LET expression, has a syntax as follows:

LET <qualification1> {':' <qualification1>}* IN <expression>

where

<qualification1> ::= <symbol>
{INSET <expression> | '|' <expression>}

In the LET expression the domain restriction is mandatory. As an example of a LET construct, suppose we have a table implemented as a function t, of one integer argument, where the key is in an even position and the value is stored

in the subsequent position. We would like to calculate "f(x) + g(x)", where x is the value corresponding to the key y in the table. We write

```
LET x ¦ EXISTS z : z >= 0 AND z MOD 2 = 0 AND t(z) = y
                   AND x = t(z+1)
    IN f(x) + g(x)
```

Note that the function t may not define a table (i.e., have more than one z whose key is y). Then the semantics is for the expression to be calculated for any x (and z) satisfying the predicate, but it is not known which ahead of time. If there is no x (or z), then the value of the expression is UNDEFINED.

A restricted form of the LET expression is called a SOME expression, and is written

```
SOME <qualification1>
```

An expression of the form

```
SOME x ¦ p(x)
```

is equivalent to

```
LET x ¦ p(x) IN x
```

The variables defined by quantified expressions and characterization expressions are called indicial variables.

H. Miscellaneous Operators

The remaining operations are the equality and inequality operations, the conditional expression, and the TYPECASE expression. Equality and inequality have already been discussed for numbers. For other types, equality and inequality will permit objects of any two non-disjoint types as arguments.

The conditional expression is of the form

```
IF b THEN e1 ELSE e2
```

where b is a BOOLEAN expression. If expressions e1 and e2 have types t1 and t2, respectively, then the type of the result is the union of t1 and t2.

Suppose we have a variable x of type "ONE_OF(t1,t2)", and wish the result of an expression to be f(x) if x is of type t1 and g(x) if x is of type t2. The most obvious solution to this problem is to provide a function "TYPE(x)" in SPECIAL to return the type of a variable x and write

```
IF TYPE(x) = t1 THEN f(x) ELSE g(x)
```

This would work in some cases, but would still produce a type error if f required an argument of type t1. The error occurs because automatic type checking cannot identify (without semantic checks) that the context of the call has restricted the type of x. Instead we provide the TYPECASE expression, which provides a context in which the automatic type checking facility can detect that an object of a united type has a particular constituent type. To solve the above problem, we write

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

The type labels (t1 and t2 in the example) must refer to disjoint types, the type of the object variable (x in the example) must be the union of all the type labels, and the type of the entire expression is the union of the types of the component expressions (f(x) and g(x) in the example).

IV. THE SPECIFICATION LEVEL

A. Introduction

In the discussion of the specification level, we discuss how to configure expressions at the assertion level in order to write module specifications.

The specification of a module is divided into six optional paragraphs so that its top-level structure looks like this:

```
MODULE  <symbol>
    TYPES
        .
        .
        .
    DECLARATIONS
        .
        .
        .
    PARAMETERS
        .
        .
        .
    DEFINITIONS
        .
        .
        .
    EXTERNALREFS
        .
        .
        .
    FUNCTIONS
        .
        .
        .
END_MODULE
```

<symbol> is the name of the module. The TYPES paragraph contains the declarations for all named types (including designators). The DECLARATIONS paragraph contains all global declarations for variables (see the next subsection). The PARAMETERS paragraph contains the declarations for symbolic constants (called <u>parameters</u>) that become bound at some time before the module is used and that cannot be changed. Module parameters are used to characterize a resource (e.g., the maximum size of a stack) or the values of initialization that are not bound to particular constants. The DEFINITIONS paragraph contains the definitions of macros whose scope is global to the module. The EXTERNALREFS paragraph contains the declarations of objects of other modules (i.e., designator types, functions with their arguments and

results, scalar types, and parameters) that are referenced in the specification. The FUNCTIONS paragraph contains the definitions (and declarations) for all the V-, O-, and OV-functions of the module.

Following a discussion of some general issues (e.g., as binding, declaration, scope rules, macros, and external references) is a description of function definition.

B. Binding, Declaration, and Scope

Every name in SPECIAL must have a binding, or a place where it is associated with a particular object. The scope of a binding is the text in a module specification over which that particular binding is in force. The scope of a binding depends on the object being bound, and may be any one of the following:

- The entire module, in the case of module parameters, function names, names for types, global macros, and constants of scalar types.
- The function definition or macro definition, in the case of formal arguments and results.
- The expression over which the variable is an index, in the case of indicial variables.

The binding of a name for which a binding is already in force is not allowed, thus eliminating the overlapping of scopes for the same name.

In most programming languages the declaration of a variable is inseparable from its binding. In SPECIAL this is the case for names of functions, macros, module parameters, and constants of scalar types, which are all bound when they are declared. However, the declaration of a variable in

SPECIAL (i.e., an indicial variable or formal argument or result) may be separated from its binding. This is done to allow a single global declaration for a variable name (in the DECLARATIONS paragraph) to apply to many different bindings. Bindings for indicial variables occur in the expressions in which they are introduced: formal arguments and results are bound either in the function definition, the external reference, or the macro definition. The motivation for global declarations is to save writing on the part of the module specifier, and to enable the establishment of mnemonics in the choice of names for globally declared variables. For example, this allows conventions such as having the variable i be of type INTEGER in all bindings. A local declaration supersedes a global one.

C. Macros

The concept of macros in SPECIAL is different from that of most programming languages. In programming languages a macro definition is generally a string substitution rule of of some complexity. The string to which the macro expands need not be a particular syntactic entity. All variables in the macro definition that are not formal arguments are bound the context of the expansion, so that the macro writer must be careful about using macro definitions with unbound variables. In SPECIAL, a macro definition has a body which is an expression and thus has a type that must be the same as the declared type of the macro definition. All macro references, which look syntactically like function references (excepting macros without arguments, which look like variables), have the same type as the declared type of the macro definition. In addition a macro definition may not use any names except its own formal arguments, and the type names and parameters of the module.

Thus, there is no chance of misusing the macro because of the context in which it is referenced.

Macros in SPECIAL are a formal shorthand (corresponding to the definitions used by a mathematician) and are not expanded as part of the syntactic processing of a module specification. They would be expanded only for proving properties based on the specification.

The syntax of a macro definition is as follows:

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

where

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

Local macros are defined within the DEFINITIONS section of a function definition, and have a scope of that function definition only. Global macros are defined in the DEFINITIONS paragraph of the entire module and have global scope.

Further work on macros might include the ability to define macros with complex type checking rules (such as the set operations of SPECIAL described above). This would probably require some restricted notion of a type variable. Also interesting would be the ability to define new types of specification language objects for which functions are part of the definition (e.g., bags as defined in [6]) and intensional constructors for these types. This kind of definitional facility would put the put the complete power of a mathematician (to define new mathematical concepts) in the hands of the specification writer, but might require so much mechanism as not to be worthwhile.

D. V-function Definition

V-functions have two purposes: to describe the state of the module and to provide information about the module's state to programs using the module. The current value of a V-function that defines a module's state is not explicitly described as part of the module specification. Instead, its current value is defined by induction: its initial value appears in its specification, and subsequent values are determined by the sequence of O-functions calls up to that point (since each O-function call relates the values of V-functions before the call to values of V-functions after the call).

A V-function may be either hidden or visible, and either primitive or derived. A hidden V-function is one that cannot be called by programs using the module, whereas a visible V-function can be. A derived V-function is one whose value is an expression derived from other V-functions of the module, whereas a primitive V-function contains part of the state definition. The form of a V-function's definition depends on its status. The syntax for a V-function definition is as follows:

```
VFUN <symbol> <formalargs> '->' <declaration> ':'
  [DEFINITIONS {,definition> ':'}+]
  [HIDDEN | EXCEPTIONS {<expression> ':'}+]
  [{ INITIALLY | DERIVATION} <expression> ':']
```

The first line, the function header, declares the formal arguments and result of the V-function. The second line declares any macros local to the function. The third line establishes whether the function is hidden or visible. If the function is hidden, the keyword "HIDDEN" appears. If the function is visible, the third line contains the keyword EXCEPTIONS followed by a list of exception conditions for the function. An exception condition is a BOOLEAN expression

which, if TRUE for a given call to the function, means that the function is not executed (thus no value is returned in the case of a V-function). Instead, control is returned to the calling program with a notification of the exception that was detected. The exceptions have an ordering, meaning that a single exception is be signalled for a given function call, even if more than one exception condition is satisfied. If the EXCEPTIONS section of a function looks like this:

```
EXCEPTIONS e1: e2: ... : en:
```

then its semantics are

```
IF e1 THEN ERRORCODE = 1
  ELSE IF e2 THEN ERRORCODE = 2
  .
  .
  .
  ELSE IF en THEN ERRORCODE = n
  ELSE UNDEFINED
```

where ERRORCODE is an abstract variable used in passing the identity of the exception beck to the calling program. Exception conditions are the same for O- and OV-functions, described in the next subsection.

If the V-function is primitive, the fourth line has the word INITIALLY followed by an assertion characterizing the function's initial value. If the V-function is derived, the fourth line has the word DERIVATION followed by an expression (of the same type as the V-function's result) that denotes its initial value.

As an example of V-function definition, consider a stack of integers maintained by a module. The state information for the module is contained in the V-functions "stack" and "ptr", signifying the elements of the stack and the stack pointer, respectively. Their definitions are as follows:

```
VFUN ptr() -> INTEGER i:
     INITIALLY i = 0:
```

```
VFUN stack(INTEGER i) -> INTEGER j:
     HIDDEN:
     INITIALLY j = ?:
```

Note that "ptr" has no exceptions, that "stack" is hidden (to prevent programs from examining intermediate values of the stack), and that the initial values of "ptr" and "stack" are 0 and UNDEFINED (or ?), respectively. The value of "ptr" signifies the number of values on the stack (also the number of integers i for which "stack(i)" is defined). We could also define a derived V-function, "top", that returns the value of the top of the stack, as follows:

```
VFUN top() -> INTEGER j:
     EXCEPTIONS ptr() = 0:
     DERIVATION stack(ptr()):
```

Note that "ptr" could also have been written as a derived V-function whose derivation would be as follows,

```
CARDINALITY({INTEGER k ¦ stack(k) ~= ?})
```

This would do away with redundancy at the expense of some clarity. Note also that asking for the top of an empty stack is meaningless, so an exception condition prevents such a call.

E. O- and OV-function Definition

The syntax of an O-function definition is as follows:

```
OFUN <symbol> <formalargs> ':'
     [DEFINITIONS { <definition> ':' }+]
     [EXCEPTIONS { <expression> ':' }+]
     [EFFECTS { <expression> ':' }+]
```

The syntax of an OV-function definition differs only in the header (the first line), which is

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

The EFFECTS section describes a state transformation by using assertions to

relate values of V-functions before the call to values of V-functions after the call. Values of V-functions before and after the call are distinguished by preceding all references to values of V-functions after the call by a single quote (´). For example, to indicate that the value of a V-function f() is incremented by an O- or OV-function call, we write

```
´f() = f() + 1
```

Note that an effect is an assertion rather than an assignment, for we can write

```
´f() > f() + 1
```

or

```
´f() + ´g() = f() + g()
```

which do not uniquely redefine the values of f() and g(). Note that all V-function values in the EFFECTS of an O-function that do not appear with a single quote are left unchanged by the O-function.

O- and OV-functions of other modules can be referenced by means of the EFFECTS_OF construct. If "o1(args)" is a reference to an O-function of another module, then the effect "EFFECTS_OF o1(args)" would expand all of the effects of "o1(args)" in the place where it was written. OV-functions may also be referenced in this way. The expression "x = EFFECTS_OF ov1(args)", where "ov1(args)" is a reference to an OV-function of another module, means that, "x is equal to the result of ov1(args) and all effects of ov1(args) are TRUE."

An example of an O-function definition is the function "push", to be used with the V-functions of the stack, described above:

```
OFUN push(INTEGER j):
     EXCEPTIONS ptr() >= maxsize;
     EFFECTS ´stack(´ptr()) = j:
          ´ptr() = ptr() + 1;
```

"maxsize" is a module parameter of type INTEGER that designates the maximum permitted stack size.

An example of an OV-function definition is provided by "pop", which pops the stack and returns the popped value, as follows:

```
OVFUN pop() -> INTEGER j:
     EXCEPTIONS ptr() = 0:
     EFFECTS j = top():
          'ptr() = ptr() - 1:
          'stack(ptr()) = ?:
```

Note that the value of j, the result of "pop", is also specified as part of the EFFECTS.

V. EXAMPLES

Table I displays the specification of a module that maintains a set of stacks as an abstract data type. Its specification differs slightly from the examples presented above, having the following interesting properties:

- A module parameter, "maxstacks", to limit the number of stacks that can exist.
- Designators of type "stack_name", to name the individual stacks, and an extra argument in each function for the stack designator.
- The functions "create_stack" and "delete_stack".
- Two global macros: "nstacks", the number of stacks that currently exist (the existence predicate for stacks is "ptr(s) ~= ?"); and "empty(s)", the empty predicate for stack s. These are both examples of mnemonics. Using mnemonics for exception conditions has been used frequently for writing larger specifications.

Note the use of global declarations: s always refers to a stack; i to a

pointer value; and j to a stack value. Note also that comments, denoted by $(...)", can be inserted anywhere in a specification.

Table II is a specification for a telephone system of a single area code. This is not a software system; it is implemented in hardware and its O-functions correspond to physical acts performed by people. However, the telephone system is a good example because everyone understands it, as opposed to most complex software systems. The system contains the following general design decisions:

- Telephones are named by a designator type (phone_id) rather than a telephone number. This corresponds to real life, in which a telephone is physically protected, i.e., knowing the number of a phone is not sufficient to be able to pick up that phone and dial from it.
- The mapping between phone numbers and connected phones is an invertible function. This is not true in a phone system with more than one area code, because a single phone number may identify different phones in different areas and because a phone is known by a different number when dialed from within the area than when dialed from outside the area.
- The state of a phone is indicated by the scalar type "phone_state". Based on the state of the phone, different things happen when an O-function is called. These states can later be mapped to particular switch positions in the actual phone circuits.
- The phenomenon that a connection can be terminated only by the party that initiated the call. Thus if the called party hangs up, the connection still exists, and the hung up phone has state "hung_up_but_connected".

There are also several features of SPECIAL whose use is worth pointing out:

- Use of a subtype to characterize a digit. Note that in this case there is no chance of misusing the subtype, since digits are only used with the equality operator.
- Use of vectors to represent phone numbers. The intensional vector constructor is used in the O-function "dial".
- Use of the SOME expression in the O-function "pick_up_phone". In this case there is only one value of "phone1" satisfying the given assertion.

The reader can see other ways of writing module specifications with the same properties as those above. The style to be chosen in writing specifications depends on one's desire for conciseness, readability, or even provability of the specifications. It is also possible to see that specifications may differ considerably from their implementations. In fact, the difference between specification and implementation is often so great that specifications cannot be construed in any way as a guide to the programmer on how to write an efficient implementation. Such information must often be supplied separately from the module specification.

VI. CONCLUSIONS

SPECIAL has been shown extremely useful for designing certain classes of systems, especially operating systems. It enables some effects of crucial design decisions to be examined at an early stage in the design process, resulting in "tight" designs for systems specified in this way. Its usefulness for proof is currently being examined. However, SPECIAL is only

one of several techniques currently in use for specifying software systems. Our experience has shown that specifications written in SPECIAL tend to be lengthy, because

- SPECIAL tries to be as general as possible, and encourages the writing of sufficient (as well as necessary) properties of software systems.
- SPECIAL tries to be both easy to read and easy to use.

More work must be done concerning tradeoffs and criteria for design of specification languages before the ultimate value of SPECIAL can be determined.

ACKNOWLEDGEMENTS

The authors are grateful to Bob Boyer, Rich Feiertag, Karl Levitt, Bernard Mont-Reynaud, J Moore, Peter Neumann, Jay Spitzen, and John Wensley for their assistance in the development of SPECIAL and the SRI Hierarchical Design Methodology.

Table I

```
MODULE stacks

    TYPES

stack_name: DESIGNATOR;

    DECLARATIONS

INTEGER i, j;
stack_name s;

    PARAMETERS
```

```
INTEGER maxsize $( maximum size of a given stack) ,
        maxstacks $( maximum number of stacks allowed) :


    DEFINITIONS

INTEGER nstacks IS CARDINALITY({ stack_name s ¦ ptr(s) ~= ? }):

BOOLEAN empty(stack_name s) IS ptr(s) = 0:

    FUNCTIONS


VFUN ptr(s) -> i:
    INITIALLY
       i = ?:

VFUN stack(s: i) -> j:
    HIDDEN:
    INITIALLY
       j = ?:

VFUN top(s) -> j:
    EXCEPTIONS
       ptr(s) = ?:
       empty(s):
    DERIVATION
       stack(s, ptr(s)):

OVFUN create_stack() -> s:
    EXCEPTIONS
       nstacks >= maxstacks:
    EFFECTS
       s = NEW(stack_name):
       ´ptr(s) = 0:

OFUN delete_stack(s):
    EXCEPTIONS
       ptr(s) = ?:
    EFFECTS
       ´ptr(s) = ?:
       FORALL i: ´stack(s, i) = ?:

OFUN push(s: j):
    EXCEPTIONS
       ptr(s) = ?:
       ptr(s) >= maxsize:
    EFFECTS
       ´stack(s, ´ptr(s)) = j:
       ´ptr(s) = ptr(s) + 1:

OVFUN pop(s) -> j:
    EXCEPTIONS
       ptr(s) = ?:
```

```
        empty(s);
    EFFECTS
        j = top(s);
        'ptr(s) = ptr(s) - 1;
        'stack(s, ptr(s)) = ?;

END_MODULE
```

Table II

```
MODULE telephone_system

    TYPES

phone_id: DESIGNATOR;
phone_state:
{ hung_up, hung_up_but_connected, dial_tone, dialing,
 dialed_unconnected_number, ringing_another_phone, being_rung, connected,
 busy } ;
digit: { INTEGER i | 0 <= i AND i <= 9 };
phone_number_id: VECTOR_OF digit;

    DECLARATIONS

phone_id phone, phone1;
digit d;
phone_number_id phone_number;
phone_state ps;

    FUNCTIONS

VFUN state(phone) -> ps: $( state of "phone")
    HIDDEN;
    INITIALLY ps = ?;

VFUN connection(phone) -> phone1:  $( "phone1" is the phone that "phone"
                                    has dialed and is either connected or
                                    ringing)
    HIDDEN;
    INITIALLY phone1 = ?;

VFUN buffer(phone) -> phone_number:  $( sequence of digits dialed
```

```
                                        by "phone")
    HIDDEN:
    INITIALLY phone_number = ?:

VFUN valid_phone_number(phone_number) -> BOOLEAN b: $(TRUE for all valid
                                                      phone numbers)
    HIDDEN:
    INITIALLY TRUE $(initialized by the phone company):

VFUN directory(phone_number) -> phone:  $( "phone_number" that
                                        corresponds to "phone")
    HIDDEN:
    INITIALLY TRUE $( initialized by the phone company) :

OVFUN install(phone_number) -> phone:  $( creates a new designator
                                       "phone" that corresponds to
                                       "phone_number")
    EXCEPTIONS
       NOT valid_phone_number(phone_number):
       directory(phone_number) ~= ?:
    EFFECTS
       phone = NEW(phone_id):
       'directory(phone_number) = phone:
       'state(phone) = hung_up:
       'buffer(phone) = VECTOR():

OFUN disconnect(phone_number):  $( disconnects phone corresponding
                                to "phone_number")
    DEFINITIONS
       phone_id phone IS directory(phone_number):
    EXCEPTIONS
       phone = ?:
       state(phone) ~= hung_up:
    EFFECTS
       'directory(phone_number) = ?:
       'state(phone) = ?:
       'buffer(phone) = ?:

OFUN pick_up_phone(phone): $( "phone" is picked up)
    EXCEPTIONS
       state(phone) = ?:
       NOT state(phone) INSET { hung_up, hung_up_but_connected, being_rung }:
    EFFECTS
       IF state(phone) = hung_up
         THEN $( picking up to dial) 'state(phone) = dial_tone
       ELSE IF state(phone) = being_rung
         THEN $( answering phone)
            'state(SOME phone1 | connection(phone1) = phone) = connected
               AND 'state(phone) = connected
       ELSE $(resuming existing connection) 'state(phone) = connected:

OFUN dial(phone: d): $( dials a digit "d" from "phone")
    DEFINITIONS
       INTEGER j IS LENGTH(buffer(phone)):
```

```
        phone_number_id newbuf
           IS VECTOR(FOR i FROM 1 TO j + 1
                     : IF i <= j THEN buffer(phone)[i] ELSE d):
        phone_id phone1 IS directory(newbuf):
    EXCEPTIONS
        state(phone) = ?:
    EFFECTS
        state(phone) INSET {dial_tone, dialing}
          => $( update buffer and change state) ´buffer(phone) = newbuf
          AND (IF phone1 ~= ?
            THEN $( a valid number has been reached) IF state(phone1) = hung_up
                THEN  $( ringing begins) ´state(phone) = ringing_another_phone
                      AND ´state(phone1) = being_rung
                     AND ´connection(phone) = phone1
                ELSE $( busy signal) ´state(phone) = busy
            ELSE IF valid_phone_number(newbuf)
              THEN $( not a connected number)
                   ´state(phone) = dialed_unconnected_number
            ELSE ´state(phone) = dialing):

OFUN hang_up(phone): $( hangs up "phone")
    EXCEPTIONS
        state(phone) = ?:
        state(phone) INSET { hung_up, being_rung, hung_up_but_connected }:
    EFFECTS
        IF EXISTS phone1 : connection(phone1) = phone
          THEN $(connection NOT terminated)
               ´state(phone) = hung_up_but_connected
        ELSE $(back to original state)
           ´state(phone) = hung_up
           AND ´buffer(phone) = VECTOR()
           AND (connection(phone) ~= ? => $(connected to someone else)
                ´connection(phone) = ?
                AND ´state(connection(phone)) =
                 (IF state(phone) = ringing_another_phone
                    THEN $(ringing stops) hung_up
                  ELSE $(terminates connection) dial_tone)):

END_MODULE
```

REFERENCES

[1] J. Guttag, E. Horowitz, and D. Musser. "The Design of Data Structure Specifications," Proc. Second International Conference on Software Engineering, San Francisco, California (October 1976).

[2] B. H. Liskov and S. Zilles. "Specification Techniques for Data Abstractions," Proc. International Conference on Reliable Software, Los Angeles, California, pp. 72-87 (April 1975).

[3] P. G. Neumann, L. Robinson, K. N. Levitt, R. S. Boyer, and A. R. Saxena. "A Provably Secure Operating System," Final Report, Project 2581, Stanford Research Institute, Menlo Park, California (June 1975).

[4] D. L. Parnas. "A Technique for Software Module Specifications with Examples," Comm. ACM 15, 5, pp. 330-336 (May 1972).

[5] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules," Comm. ACM 15, 12, pp. 1053-1058 (December 1972).

[6] L. Robinson and K. N. Levitt. "Proof Techniques for Hierarchically Structured Programs," Technical Report CSL-27, Computer Science Laboratory, Stanford Research Institute, Menlo Park, California, to appear in Comm. ACM (November 1975).

[7] L. Robinson, K. N. Levitt, P. G. Neumann, and A. R. Saxena. "On Attaining Reliable Software for a Secure Operating System," Proc. International Conference on Reliable Software, Los Angeles, California, pp. 267-284 (April 1975).

[8] O. M. Roubine and L. Robinson. "SPECIAL Reference Manual," Technical Report CSL-45, Computer Science Laboratory, Stanford Research Institute, Menlo Park, California (August 1976).

[9] J. M. Spitzen, K. N. Levitt, and L. Robinson. "An Example of Hierarchical Design and Proof," Technical Report CSL-30, Computer

Science Laboratory, Stanford Research Institute, Menlo Park, California (also submitted for publication) (March 1976).

[10] R. S. Boyer and J S. Moore. "Proving Theorems About LISP Functions," J. ACM 22, 1 pp. 129-144 (January 1976).

[11] J S. Moore. "The INTERLISP Virtual Machine Specification," Technical Report, Computer Science Laboratory, Xerox Palo Alto Research Center, Palo Alto, California (September 1976).

[12] C. A. R. Hoare and N. Wirth. "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica 2, pp. 335-355 (1973).

APPENDIX: GRAMMAR OF SPECIAL

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

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

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


<delay>                 ::= DELAY UNTIL <expression> ':'

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

<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> ')'
                        ::= NEW '(' <symbol> ')'
                        ::= [EFFECTS_OF] <call>
                        ::= <structureconstructor>
                        ::= <vectorconstructor>
                        ::= <setexpression>

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

<qualification>         ::= [<typespecification>] <symbol> { '|' | 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> '}'
```

DISTRIBUTION LIST

Mr. Tony Allos Code 6201 — 1 copy
Naval Electronics Laboratory Center
271 Catalina Boulevard
San Diego, CA 92151

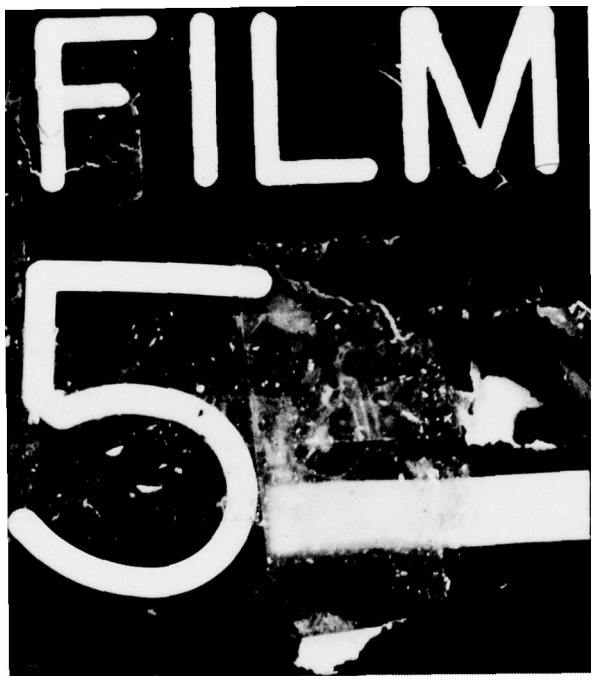
Mr. L. Sutton Code 5200 — 35 copies
Naval Electronics Laboratory Center
271 Catalina Boulevard
San Diego, CA 92151

Mr. William Carlson — 15 copies
Advanced Research Projects Agency
Office of Secretary of Defense
1400 Wilson Boulevard
Arlington, VA 22209

Mr. Neal Hampton Code 5200 — 15 copies
Naval Electronics Laboratory Center
271 Catalina Boulevard
San Diego, CA 92151

Professor Stuart Madnick — 1 copy
MIT - Sloan School
E53-330
Cambridge, Mass 02139

Mr. John Machado — 5 copies
The Naval Electronic Systems Command
National Center No. 1
Crystal City, Washington, D.C. 20360


FILM
5