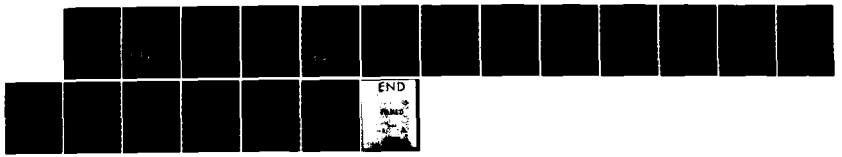
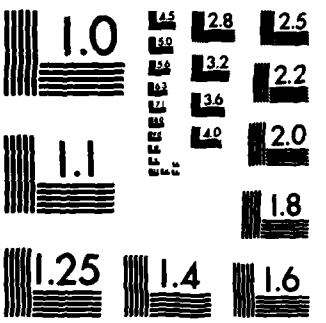
AD-A143 707 A STANDARD ORGANIZATION FOR SPECIFYING ABSTRACT INTERFACES(U) NAVAL RESEARCH LAB WASHINGTON DC P C CLEMENTS ET AL. 14 JUN 84 NRL-8815 SBI-AD-E000 582 1/1

UNCLASSIFIED F/G 9/2 NL



END



NRL Report 8815

A Standard Organization for Specifying Abstract Interfaces

PAUL C. CLEMENTS, R. ALAN PARKER,
DAVID L. PARNAS,* AND JOHN SHORE

Computer Sciences and Systems Branch
Information Technology Division

**Also at University of Victoria,*
Victoria, B.C.

KATHRYN H. BRITTON

IBM
Research Triangle Park, North Carolina

June 14, 1984

NAVAL RESEARCH LABORATORY
Washington, D.C.

SECURITY CLASSIFICATION OF THIS PAGE

AD-A143 757

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS N/A
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) NRL Report 8815	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Naval Research Laboratory 6b OFFICE SYMBOL (If applicable) Code 7590	7a NAME OF MONITORING ORGANIZATION
6c ADDRESS (City, State and ZIP Code) Washington, DC 20375	7b ADDRESS (City, State and ZIP Code)
8a NAME OF FUNDING/SPONSORING ORGANIZATION (See Page ii) 8b OFFICE SYMBOL (If applicable)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c ADDRESS (City, State and ZIP Code) Washington, DC 20360	10 SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO: 62721N; PROJECT NO:; TASK NO: SF21243601; WORK UNIT NO: DN 980087
11 TITLE (Include Security Classification) (See Page ii)	
12 PERSONAL AUTHOR(S) Clements, Paul C.; Parker, R. Alan; Parnas, David L.*; Shore, John E.; and Britton, Kathryn H.†	
13a TYPE OF REPORT Interim 13b TIME COVERED FROM N/A TO	14 DATE OF REPORT (Yr., Mo., Day) 1984 June 14 15 PAGE COUNT 19
16 SUPPLEMENTARY NOTATION *Also at University of Victoria, Victoria, B.C. †IBM, Research Triangle Park, North Carolina	
17 COSATI CODES FIELD / GROUP / SUB GR	18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Software specifications; Abstract interfaces; Software engineering; Software documentation

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

NRL's Software Cost Reduction project is demonstrating the feasibility of applying advanced software engineering techniques to complex real-time systems to simplify maintenance. To demonstrate the principles, the onboard software for the Navy's A-7E aircraft is being redesigned and reimplemented. The project is producing a set of model procedures and documents that can be followed by designers and producers of other software systems.

This document describes the format to be followed in documenting the interfaces of the software modules.

(Continued)

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS ☐	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Paul C. Clements	22b TELEPHONE NUMBER (Include Area Code) 202-767-3477 22c OFFICE SYMBOL Code 7595

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE

8a. NAME OF FUNDING/SPONSORING ORGANIZATION

Naval Electronics Systems Command

11. TITLE *(Include Security Classification)* (Continued)

A Standard Organization for Specifying Abstract Interfaces

19. ABSTRACT (Continued)

An abstract interface is a software module interface that remains constant, even when details of the software implementation change. Specifying such interfaces is a key to designing software systems for change. The format described in this report is designed to serve the author who designs a module, the coder who implements it, designers of other modules that must make use of it, and reviewers who must approve its design. It organizes the specification into a small number of concise, well-defined sections, allowing readers who are searching for a particular kind of information to look in a particular section. All module interface descriptions of NRL's Software Cost Reduction project use this format.

CONTENTS

A STANDARD ORGANIZATION FOR SPECIFYING ABSTRACT INTERFACES

1. INTRODUCTION

There are three major tasks in designing a software system. The first is partitioning the system into work assignments (modules). The second is designing the interface of each module, i.e., decide what facilities the module will provide. The third is producing the specification for each interface so that (a) the implementers have enough information to write the software; (b) writers of other modules have enough information to use the module; and (c) information that constrains or discloses details of the implementation is not revealed.

NRL's Software Cost Reduction (SCR) project is a program that is investigating new approaches to these and other software engineering problems by testing the feasibility of applying modern software engineering techniques (such as information-hiding modules) to demanding software environments, such as an embedded real-time avionics system. Information-hiding is the approach taken in the first two tasks; this document explains the approach taken to solve the third.

Information-hiding [1] is a method of designing software to minimize the impact (and hence, the cost) of making software changes. The method involves dividing the software into modules according to likely changes; each module is responsible for encapsulating or "hiding" the effects of a change from the rest of the system. The key is to design the interface of each module so that it consists only of information about that particular module that is *not* likely to change. In that way, when changes that affect a module are required, only the implementation of that module is likely to require a change. The interface and all other modules that use the interface are not likely to change at all.

This interface is called an *abstract interface,* because it represents an *abstraction* of the entire module, in the same way that a road map of the world is an abstraction of the world. There are many things about the world that could change (e.g., the location and number of all the buildings, trees, people, etc.), but a user of a road map is not concerned with these things; hence, the map does not necessarily change.

The A-7E Software Module Guide [2] documents the decomposition into modules of the SCR software and explains how the information-hiding principle was applied to achieve the modularization and the resulting interface designs.

To meet the goals for module interface specifications set forth in the first paragraph, the specification of an abstract interface should have the following properties:

- It must not disclose any of the changeable aspects ("secrets") of the module;
- It must present a concise description of the facilities available from the module, in terms of effects that are observable to the user;

Manuscript approved January 13, 1984.

- It should be divided into sections and formatted so that a reader unfamiliar with the module is able to find a piece of information without having to study the entire interface specification; i.e., it should serve the quick-reference reader as well as the first-time reader;
- It should not provide duplication of information, which would make using and maintaining the document more difficult.

The organization chosen to achieve these properties consists of the following sections:

Introduction	A brief prose overview of the module's facilities to help the reader determine if this is the module he is interested in;
Interface Overview	A table of programs on the module's interface, showing the parameters and parameter types and stating the effects of each;
Local Type Definitions	Definitions of the data types available to users from the module;
Dictionary	Definitions of any specialized terms used throughout the specification;
Undesired Event Dictionary	Definitions of any possible incorrect uses (errors) of the module's facilities;
System Generation Parameters	A list of those quantitative characteristics of the module that are not bound until just before run-time (e.g., the size of a data structure);
Facilities Index	A quick look-up reference of all programs and terms defined in the specification;
Design Issues	A prose section explaining why certain design decisions were made to aid people who might make future changes to the design;
Implementation Notes	A prose section to capture information that might have come to the designer's attention that would be of use to the implementors;
Assumptions Lists	A prose section documenting the assumptions that the users of the module are allowed to make about it.

Complete descriptions of each section follow.

2. DESCRIPTION OF THE STANDARD ORGANIZATION

The format for specifying an abstract interface consists of the following sections:

2.1 Introduction

This section introduces, in informal prose, the features provided by the module. It may define basic concepts that are used in the rest of the specification.

2.2 Interface Overview

The interface overview section includes tables that provide an overview of facilities provided by the module. Facilities generally fall into two categories—access programs that users may call, and events that the module reports and that users may await. Readers familiar with the module interface can use these tables to refresh their memories about particular facts without having to reread the longer explanations in later sections. The interface overview may contain any of the following subsections:

2.2.1 Access Program Table

Figure 1 shows the form for the access program table. This table lists all access programs provided by the module, as well as the number, data type, and semantics of the parameters. Access programs can change or retrieve information that is stored in a module's internal storage. Access program names begin and end with brackets that show when they can be used: ++ brackets indicate programs that may be invoked only at system-generation time; + brackets enclose programs that are executable at run time. The access program table contains an entry for each access program provided by the module; each entry includes the program name, parameter data, and undesired events (discussed more fully later) associated with the program.

There are three types of access programs. Each type is characterized by the facilities offered to user programs, the effects on other access programs provided by the module, the information required to specify them, and the naming conventions.

Value Programs—These programs deliver values to user programs via output parameters. A call to a value program has no effect on subsequent calls to that program or on any other program of the same module. Semantics of value programs are given in the dictionary definition(s) of the term(s) used to denote the output parameter(s). Value program names usually begin with G_ for Get value.

Effect Programs—These programs enable user programs to affect the future operation of the module by passing it information or giving it commands. Effect programs may affect the values returned by subsequent calls to value programs, may change the values shown by display devices, or may affect the current operating state of the module. These programs do not return values themselves. The parameter-information column in the access program table can be left blank for these programs whenever the program effects section adequately defines the parameter meanings. The names of effect programs usually begin with S_ for Set value.

Hybrid Programs—These programs have characteristics of both value and effect programs: they return values and affect the future operation of the module. These programs will usually be described both by parameter-information entries in the access program table and descriptions in the effects section.

Matching Value/Effect Program Pairs—In some cases, value and effect programs are matched so that the value program always returns the most recent value set by the effect program. For the sake of clarity and brevity, these programs are described together. Matching program pairs always share the same name, except that the value program starts with G_ and the effect program starts with S_. These programs are defined in a single line in the access program table, with the program names given together as +G/S_. For instance, +G/S_ADC_LPROBE+ actually refers to two programs: +G_ADC_LPROBE+ and +S_ADC_LPROBE+.

In the SCR software environment, the exact syntax for invoking an access program is given in Ch. EC. PGM of Ref. 3.

Program Name	Parm type	Parm info	Undesired Events
++program1++ or +program1+	p1:type1;K	info1	%%name1%% or %name1%
	p2:type2;K	info2	%%name2%% or %name2%
	.	.	.
	.	.	.
	pN1:typeN1;K	infoN1	%%nameM%% or %nameM%
++program2++ or +program2+	p1:type1;K	info1	
	p2:type2;K	info2	
	.	.	
	.	.	
	pN2:typeN2;K	infoN2	
.			
.			
.			
++programG++ or +programG+	p1:type1;K	info1	
	p2:type2;K	info2	
	.	.	
	.	.	
	pN2:typeNG;K	infoNG	

Fig. 1 – Access program table format

Legend for Fig. 1

Underscored symbols are *required* but without the underscores. Other names and letters are defined as follows:

G	number of programs in the group, where group is defined as a set of programs with the same entries in the undesired events column; different groups are separated by a horizontal line in the table.
programJ	name of the Jth program in the group, where J = 1,...,G. If the name contains ++ brackets, that program may only be invoked at system-generation time; that is, that program will exist only in the support software prior to the time the software is loaded onto the target machine. A name with + brackets may be invoked at run-time; that is, that program will be available for invocation on the target machine.
NJ	number of parameters for the Jth program. If zero, the parameter columns are empty for the program.
pL	the Lth parameter of a program, where L = 1,... NJ
typeL	type of parameter pL: the name of a data type provided either by this module or another. If provided by this module, it will be defined in the Local Types section of the specification.
K	I, O, IO for input, output, and input-output parameter. Programs receive the values of input parameters and deliver the values of output parameters. Input-output parameters serve both purposes. Parameters are separated by commas in the call statement.
infoL	definition of the meaning of parameter pL; may be an entry in the specification's dictionary (!+entry L+!) or an expression involving other parameters, such as p1 + p2; infoL may be omitted for any parameter whose meaning is given in the effects section, or it may be an informal description summarizing the program effect description.
M	number of UE dictionary entries defined for the group
nameE	Entry E in the specification's UE dictionary, where E = 1,2,...M; the dictionary entry defines the circumstances that cause a program call to be illegal. If the name contains %% brackets, the UE will be detected by the module before run-time, and the user may not provide a run-time program to handle the UE. If the name has % brackets, the UE may not be detected until run-time, and the user is obligated to provide a run-time UE-handling program for it. Naturally, system-generation-time programs can only have system-generation-time UEs associated with them.

2.2.2 Events

This section is a table that contains a list of all of the events reported by the module. Events are reported via access programs that do not return until the specified conditions hold. There are four varieties of event-reporting programs:

@Tcondition This program will return when condition next changes from false to true.

@Fcondition This program will return when condition next changes from true to false.

=Tcondition This program will return when condition is true, as soon as the applicable process synchronization rules permit (in the SCR software, these are documented in Ch. EC.PAR of Ref. 3).

=Fcondition This program will return when condition is false, as soon as the applicable process synchronization rules permit (in the SCR software, these are documented in Ch. EC.PAR of Ref. 3).

Condition is an entry in the module specification's dictionary (see Section 2.4). Event meanings are thus defined by the associated dictionary entries.

Because one condition may correspond to four event programs with similar semantics, a shorthand has been adopted that combines the names of the possible programs. For example, let *x* be a condition. Then the string @T/@F/=T/=Fx names four programs: @Tx, @Fx, =Tx, and =Fx, each with semantics for condition *x* as described above.

2.2.3 Effects

This section specifies the effects (semantics) of invoking a hybrid or effect access program. The effects are specified completely in terms of changes or results that are completely observable by using software or a human observer. It is basic to the information-hiding methodology that no information about the implementation or other hidden aspects of a module be divulged in this section. Effects may be given by specifying changes in the values that will subsequently be returned by access programs, or in terms of events that will occur at a later time. An example of a human-observable effect is the positioning of a symbol on a display. If any run-time undesired events are enabled or disabled as a result of invoking the program, that is also described here.

2.3 Local Data Types

For every program parameter, a type is specified in the interface overview. This section of the specification defines the data types that are used in communicating with the module. All such data types are described in this section except those that are defined in another module interface specification, in which case a reference to that specification is to be given. Some data types are called "enumerated types"; these are described by a list of strings or a syntax that defines the list of strings eligible to be passed to the program.

2.4 Dictionary

This section of the specification defines terms that appear using the !+term+! and !!term!! notation in other sections of the specification.

An item of the form !+term+! is used in the access program table to name an output parameter of a program. The dictionary definition of such a !+term+!, then defines the value returned by the

access program via the output parameter. This gives the semantics of the program. As in program effects, the definition is given only in terms that can be tested by the software or a human user. A !+term+! may also be imbedded in the name of an event-reporting program, and the definition of the term thus defines the semantics of the event that is reported by the program.

A !!term!! may be used *anywhere in the specification* (except to describe an output parameter of a program) to take the place of a specialized technical definition that would otherwise have to be repeated.

The definitions are prose, given in alphabetical order by term.

2.5 Undesired Event Dictionary

An undesired event (UE) occurs when an assumption about an undesired events is violated, usually when an access program is called with an incorrect parameter or in a state in which it cannot be executed successfully. This section defines the conditions that correspond to each undesired event reported by the module.

A UE is considered *enabled* when the UE may occur and *inhibited* when it cannot occur. Some UEs are always enabled. Some UEs are inhibited or enabled by access programs (user-controlled state UEs). Some UEs are inhibited or enabled by changes detected within the module (internal state UEs), and their status is available via access programs.

This section defines the %term% or %%term%% entries in the access program table by stating the violation that each one represents. A UE of the form %%term%% will be detected at system generation time. A UE of the form %term% may not be detected until run time. The specification describes user-controlled state UEs in terms of the commands that inhibit or enable them and internal state UEs in terms of the value programs that reveal whether the UE is currently inhibited or enabled.

2.6 System Generation Parameters

This section describes those externally visible characteristics of the module that can be changed by assigning values to parameters at system generation time. Each parameter is named, its data type is given, and its meaning is described. These parameters are denoted by #term#, and may be used as symbolic constants by users of the module.

2.7 Facilities Index

After all the submodules in the document have been specified using the foregoing scheme, an index is provided that shows where in the document a particular name is defined. The index includes a list of access programs, instructions, local data types, dictionary items, undesired event names, and system generation parameters. The system generation parameter list includes a range of expected values for each parameter.

2.8 Interface Design Issues

This prose section describes any alternative designs that were considered and records the reason for their rejection. The section serves as a history of design decisions, so that issues are not considered repeatedly. It serves as a design rationale providing guidance to maintenance programmers revising the program.

2.9 Implementation Notes

This prose section contains implementation notes. During the design of the module interface, certain facts or ideas may come to the designer's attention, ideas that would be necessary or useful to future implementers, and these are noted in this section. As the module is implemented, the section may be deleted, moving the information into the module implementation documentation.

2.10 Assumptions Lists

The information in the assumptions lists is redundant. It is implied by the description of the facilities specified in the rest of the section. The purpose of the assumption list is to serve as an explicit medium for review by nonprogrammers.

This section comprises two prose subsections.

2.10.1 Basic Assumptions

These assumptions contain information that users of the module may assume will never change. In the case of hardware-hiding modules [2], it consists of information that will remain true about the interface even if the hidden hardware is replaced or modified. In the case of requirements-hiding modules, it consists of information that will remain true even if the hidden requirements are changed. In the case of software decision-hiding modules, it consists of information that will remain true even if the hidden software decisions are changed.

The assumptions relate to the normal use and operation of the module. A basic assumption will fall into one of two categories: implementability (an assumption that the module's facilities can be implemented efficiently), and sufficiency (an assumption that the given facilities are all the user will ever need). Specifically, they may concern: (a) information available from the module; (b) information that must be supplied to the module; (c) events that can be reported by the module; (d) tasks that can be performed by the module; (e) operating states of the module and how they affect the information available and the information required; or (e) failure states of the module and how they affect the information available.

2.10.2 Assumptions About Undesired Events

This section lists assumptions describing *incorrect* usage of the module at run-time. Violation of each assumption is associated with a run-time undesired event. The development version of the system will be designed to report the undesired event whenever a violation occurs. In the production version of the system, the undesired event-handling code will be removed, and violations of the assumptions in this section will result in unpredictable behavior.

3. NOTATION CONVENTIONS

The following table lists the notational brackets used and indicates what section(s) of an interface specification gives relevant information.

Notation	Meaning	Where to Look It Up
++name++	A module access program that may only be invoked at system generation time	Section 2
+name+	A module access program that may be invoked at run-time	Section 2
+G_name+ or ++G_name++	A value access program; does not change the state of the module, but returns a value described in the dictionary	Sections 2,5
+S_name+ or ++S_name++	An effect access program; changes the module state as described in Section 2. *Usually returns no value.*	Section 2
%%name%%	An undesired event that will be detected at system-generation time	Sections 2,5
%name%	An undesired event that may not be detected until run-time	Sections 2,5
!+name+!	Either the name of a value produced by a module's access program, or the name of a condition associated with an event; its definition is given in the specification's dictionary section.	Sections 2,4
!!name!!	Used to denote a term with a specialized definition that appears frequently in the specification; its definition is given in the specification's dictionary section.	Section 4
@Tname @Fname =Tname =Fname	*The name of an access program that will* not return until the associated condition is satisfied or the associated event occurs	Section 2
#name#	The name of a system-generation time parameter	Section 6

4. EXAMPLE

The following is an example to illustrate the form of the first six sections of a specification of an abstract interface. (The remaining sections of the specification, composed of an index and prose paragraphs, are not shown.) This abbreviated example is drawn from Ref. 3; the submodule specified provides data declaration and manipulation facilities for an abstract computer.

EC.DATA
DATA MANIPULATION FACILITIES

EC.DATA.1 INTRODUCTION

The Extended Computer Data module provides literals, constants, and variables. We refer to these as *entities. Literals* are values appearing in programs. *Constants* have names and values; run-time programs can read the values but not change them. *Variables* have names and values; the values can be read or written by run-time programs.

Types are classes of entities. This module provides the real and bitstring type classes. Specific real types are characterized by !!range!! and !!resolution!!. Specific bitstring types are characterized by length. There may exist any number of specific types. The attributes of a type may not be changed once declared.

EC.DATA.2 INTERFACE OVERVIEW

EC.DATA.2.1 ACCESS PROGRAM TABLE – DECLARING SPECIFIC TYPES AND ENTITIES

Program name	Parm type	Parm info	Undesired events
++DCL_TYPE++	p1:name;I p2:typeclass;I p3:attribute;I	Name of new type Containing type class Attributes of type	%%name in use%% %%inappropriate attributes%% %%length too great%% %%range too great%% %%res too fine%%

Program Effects

A specific type that is a member of type class p2 is declared to have identifier p1. All entities of this specific type will have the attributes given by p3. The identifier can be used as the spectype (p2) parameter in calls on ++DCL_ENTITY++ in programs that follow the declaration.

Program name	Parm type	Parm info	Undesired events
++DCL_ENTITY++	p1:name;I p2:spectype;I p3:convar;I p4:constant or literal whose value is in domain of type named by p2;I	Entity name Specific type of entity When writable Initial value	%%name is use%% %%undeclared spectype%% %%unknown initial value%% %%wrong init value type%% %%value too big%%

Program Effects

An entity with identifier p1, spectype p2, and initial value p4 is declared. If p3=VAR, the entity may be used as a !!destination!! in a subsequent operation. The entities that have been declared may be used as operands in the programs that follow.

EC.DATA.2.5 ACCESS PROGRAM TABLE – OPERATIONS ON REAL ENTITIES

Program name	Parm type	Parm info	Undesired events
+EQ+ +NEQ+ +GT+ +GEQ+ +LT+ +LEQ+	p1:real;I p2:real;I p3:boolean;0 p4:real;I	!!source!! !!source!! !+destination+! !!user threshold!!	%%constant destination%%

Program name	Parm type	Parm info	Undesired events
+ADD+	p1:real;I	!!source!!	%%constant destination%%
+MUL+	p2:real;I	!!source!!	%range exceeded%
+SUB+	p3:real;0	!+destination+!	
+SET+	p1:real;I	!!source!!	
++SET++	p2:real;0	!+destination+!	
+DIV+	p1:real;I	!!source!!	%range exceeded%
	p2:real;I	!!source!!	%divide by zero%
	p3:real;I	!+destination+!	%%constant destination%%

Program Effects

+ADD+	p3 = p1 + p2
+EQ+	p3 = (p1 = p2)*
+GEQ+	p3 = (p1 = p2)* OR (p1 − p2 is positive)
+GT+	p3 = p1 − p2 is positive and NOT (p1 = p2)*
+LEQ+	p3 = (p1 = p2)* OR (p1 − p2 is negative)
+LT+	p3 = p1 − p2 is negative and NOT (p1 = p2)*
+MUL+	p3 = p1 * p2
+NEQ+	p3 = NOT (p1 = p2)*
+SET+	p2 = the value of p1 before the operation
++SET++	p2 = the value of p1 before the operation
+SUB+	p3 = p1 − p2
+DIV+	p3 = p1/p2 This is the slowest divide program available on the EC.

*Definition of equality (=):

absv(p1 − p2) is less than or equal to threshold, where threshold is MAX(p4, 1/2 * MAX(!!resolution!!(p1), !!resolution!!(p2))).

EC.DATA.2.7 ACCESS PROGRAM TABLE — OPERATIONS ON BITSTRING ENTITIES

Program name	Parm type	Parm info	Undesired events
+AND+	p1:bitstring;I	!!source!!	%inconsistent lengths%
+OR+	p2:bitstring;I	!!source!!	%%constant destination%%
+XOR+	p3:bitstring;O	!+destination+!	
+NOT+	p1:bitstring;I	!!source!!	
	p2:bitstring;O	!+destination+!	
+SHIFT+	p1:bitstring;I	!!source!!	
	p2:integer;I	shift length	
	p3:bitstring;O	!+destination+!	

Program Effects

+AND+	p3 = p1 AND p2
+NOT+	p2 = NOT p1
+OR+	p3 = p1 OR p2
+SHIFT+	p3 = shift of p1 by p2 positions to the right (or −p2 positions to the left). The vacated bits are set to O:B.
+XOR+	p3 = (p1 AND (NOT p2)) OR (p2 AND (NOT p1))

EC.DATA.3 LOCAL TYPE DEFINITIONS

attribute An attribute for a bitstring is a positive integer specifying length.
A real attribute is a parenthesized list:
(lower bound, upper bound, !!resolution!!)
The lower bound and upper bound are often collectively called range (see !!range!!).

bitstring An ordered list of values, each value represented by 0 or 1. The number of such values is called the *length* of the bitstring. Bits in all bitstring types are numbered from 0 upward. We refer to bit 0 as the leftmost bit and a shift of information from higher numbered bits to lower numbered bits as a left shift. A bitstring literal is written as a string of 0s and 1s suffixed by :B, e.g.,

0:B bitstring of length 1
1011:B bitstring of length 4

boolean Bitstring of length 1. Where convenient, $true$ may denote 1:B, $false$ may denote 0:B.

convar Either ASCON (meaning constant that will not change without a reassembly) or LOADCON (meaning constant that may be changed by a memory-loading device while the program is not running) or VAR (meaning variable).

integer Real with !!resolution!! = 1.

name An identifier for an object created. A name must consist only of alphanumerics or one of the following: +#%@/$( )_

real An approximation to conventional real numbers. Real literals are denoted in the standard decimal notation format, e.g., 112.345, .000234, 127

spectype An identifier that has been previously declared as a type in a ++DCL_TYPE++ operation, or the identifier BOOLEAN, representing the built-in spectype boolean.

typeclass Either BITS (meaning bitstring) or REAL.

EC.DATA.4 DICTIONARY

Term	Definition
!+destination+!	A variable that will contain results of operation.
!!destination!!	An O or IO actual parameter to an EC access program or a user-defined EC program.
!!range!!	The set of values between (and including) the lower bound and upper bound of a real-data type.
!!resolution!!	The maximum difference between any two consecutive representatives of the values of a real data type.
!!source!!	An I or IO actual parameter to an EC access program.
!!user threshold!!	A difference that user programs specify for a comparison operation; i.e., two numbers whose difference is less than this are considered equal.

EC.DATA.5 UNDESIRED EVENT DICTIONARY

%%constant destination%%	A user program attempted to use a constant entity as a !!destination!!.
%divide by zero%	A user program attempted to divide by zero.
%%inappropriate attributes%%	The attributes given are not valid for the type class at hand.
%inconsistent lengths%	The length of the result of a bitstring operation differs from the length of the destination variable.
%%length too great%%	The length of a bitstring type exceeds the maximum allowed.
%%name in use%%	An attempt has been made to redefine a name of one of the following: an EC access program, an EC UE, and EC system generation parameter, or a user-defined spectype, entity.
%range exceeded%	The value being stored into a variable is outside the !!range!! of the variable.
%%range too great%%	The magnitude of the declared !!range!! exceeds the maximum allowed for that typeclass, as given by a system generation parameter.
%%res too fine%%	Declared resolution (or implied resolution of a literal) was less than the minumum allowed for that typeclass, as given by a system generation parameter.
%%undeclared spectype%%	The user has supplied an undeclared spectype in an *entity declaration.*
%%unknown initial value%%	A variable has been used as an initial value of a declared entity.
%%value too big%%	The value of a real entity is greater in magnitude than that allowed, as given by a system generation parameter.
%%wrong init value type%%	A constant or literal used as an initial value is not in the domain of the type of the entity being initialized.

EC.DATA.6 SYSTEM GENERATION PARAMETERS

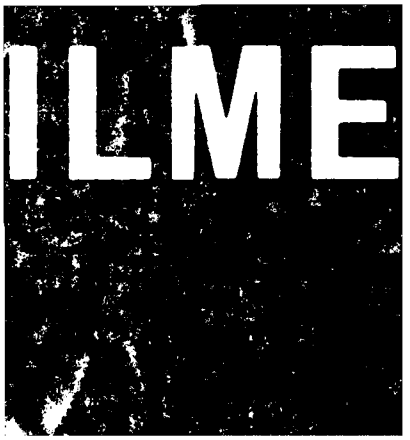
Parameter	Type	Definition
#max bits length#	integer	The maximum number of bits allowed for a bitstring.
#max real size#	real	Maximum allowable magnitude for a real ascon or literal.
#max real range#	real	Maximum allowable magnitude for the absv (upper bound - lower bound) for a real type.
#min real resolution#	real	Minimum allowable resolution for a real entity.

REFERENCES

1. D.L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," *Comm. ACM*, 15 (12), 1053-1058 (1972).

2. K.H. Britton and D.L. Parnas, "A-7E Software Module Guide," NRL Memorandum Report 4702, Dec. 1981.

3. K.H. Britton, P.C. Clements, D.L. Parnas, and D.M. Weiss, "Interface Specifications for the (SCR) A-7E Extended Computer Module," NRL Report 4843, May 1982.


ILME