INTERNATIONAL STANDARD

ISO/IEC 13886

First edition
1996-03-15

Information technology — Language-Independent Procedure Calling (LIPC)

Technologies de l'information — Appel de procédure indépendant du langage (LIPC)



Reference number
ISO/IEC 13886:1996(E)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS

Not for Resale

Contents

1 Scope 1

2 References 1

3 Definitions and Abbreviations 2
3.1 Definitions . . . 2
3.2 Abbreviations . . . 4

4 Conformance 5
4.1 Modes of conformance . . . 5
4.1.1 Client mode conformance . . . 5
4.1.2 Server mode conformance . . . 5

5 A model of procedure calling: informal description 6
5.1 Model overview . . . 6
5.2 Parameter passing . . . 7
5.2.1 Methods of parameter passing . . . 9
5.2.1.1 Call by Value Sent on Initiation . . . 9
5.2.1.2 Call by Value Sent on Request . . . 9
5.2.1.3 Call by Value Returned on Termination . . . 10
5.2.1.4 Call by Value Returned when Available . . . 10
5.2.2 Global data . . . 10
5.2.3 Parameter Marshalling / Unmarshalling . . . 11
5.2.4 Pointer Parameters . . . 11
5.2.5 Private types . . . 12
5.3 Execution-time Control . . . 12
5.3.1 Terminations . . . 12
5.3.1.1 Normal termination . . . 12
5.3.1.2 Abnormal Termination . . . 13
5.3.1.3 External Cancellation . . . 13
5.3.1.4 Predefined conditions . . . 13
5.4 Execution Control . . . 14
5.4.1 Synchronous and Asynchronous Calling . . . 14
5.4.2 Recursion . . . 14

6 A model of procedure calling: formal description 14
6.1 Value . . . 14

© ISO/IEC 1996
All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

6.2 Boxes and global state . . . 14
6.3 Symbol . . . 15
6.4 Procedure image . . . 16
6.5 Association . . . 16
6.6 Procedure closures . . . 17
6.7 Boxes, pointers, values, and datatypes . . . 17
6.8 Interface closure . . . 18
6.9 Interface type . . . 19
6.10 Specifications . . . 19
6.11 Basic procedure invocation . . . 20
6.12 Type correctness . . . 20
6.13 Associates . . . 21
6.13.1 Simple Associates . . . 21
6.13.2 Generalized Associates . . . 22
6.14 Execution and Invocation contexts . . . 23
6.15 Parameter translations . . . 24
6.16 Defining Translation Procedures . . . 26

7 Interface Definition Notation **27**
7.1 Definitional Conventions . . . 27
7.1.1 Character Set . . . 27
7.1.2 Formal Syntax . . . 27
7.1.3 Whitespace . . . 28
7.2 Interface Type Declarations . . . 29
7.2.1 Type references . . . 29
7.2.2 Value References . . . 30
7.3 Import Declarations . . . 30
7.4 Value Declarations . . . 31
7.5 Datatype Declarations . . . 31
7.5.1 Primitive Datatypes . . . 32
7.5.1.1 Integer . . . 32
7.5.1.2 The real datatype . . . 32
7.5.1.3 The character datatype . . . 33
7.5.1.4 The boolean datatype . . . 33
7.5.1.5 The enumerated datatype . . . 33
7.5.1.6 The octet datatype . . . 33
7.5.1.7 The procedure datatype . . . 34
7.5.1.8 The state datatype . . . 36
7.5.1.9 The ordinal datatype . . . 36
7.5.1.10 The time datatype . . . 37
7.5.1.11 The bit datatype . . . 37
7.5.1.12 The rational datatype . . . 37
7.5.1.13 The scaled datatype . . . 37
7.5.1.14 The complex datatype . . . 38
7.5.1.15 The void datatype . . . 38
7.5.2 Generated datatypes . . . 38
7.5.2.1 The record datatype . . . 38
7.5.2.2 The choice datatype . . . 39

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

7.5.2.3 The array datatype . . . 39
7.5.2.4 The pointer datatype . . . 40
7.5.3 Subtypes . . . 40
7.6 Parameterized Types . . . 40
7.7 Identifiers . . . 41
7.7.1 Value references to fields . . . 42
7.7.2 Value references to parameters, return-args, or to fields contained within them 42
7.7.3 Value references to formal-value-parms . . . 43
7.7.4 Value references to value-expressions . . . 43
7.7.5 Value references to enumeration-identifiers . . . 43
7.7.6 Termination references . . . 44

Annexes

A Procedure Parameters 45
A.1 LIPC Reference / Local Access . . . 45
A.2 LIPC Reference / LIPC Access . . . 45
A.3 Local Reference / Local Access . . . 46

B Interface Definition Notation Syntax 47

C How to do an LIPC binding for a language 53
C.1 Linking the client and the server . . . 53
C.2 Client mode binding . . . 54
C.3 Server mode binding . . . 55
C.4 Procedure parameters . . . 56
C.5 Global variables . . . 57

D LIPC IDN - RPC IDL Alignment overview 58
D.1 Interface Declarations . . . 58
D.1.1 Attributes . . . 58
D.1.2 Imports Clause . . . 58
D.2 Other Declarations . . . 59
D.2.1 Type Declarations . . . 59
D.2.2 Value Declarations . . . 59
D.2.3 Procedure Declarations . . . 59
D.2.4 Termination Declarations . . . 60
D.3 Primitive Datatypes . . . 60
D.3.1 Boolean . . . 60
D.3.2 State . . . 60
D.3.3 Enumerated . . . 60
D.3.4 Character . . . 61
D.3.5 Ordinal . . . 61
D.3.6 Time . . . 61
D.3.7 Integer . . . 61
D.3.8 Rational . . . 62
D.3.9 Scaled . . . 62
D.3.10 Real . . . 62

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.3.11 Complex . . . 63
D.3.12 Void . . . 63
D.4 Type Qualifiers . . . 63
D.5 Generated Datatypes . . . 63
D.5.1 Choice . . . 63
D.5.2 Pointer . . . 64
D.5.3 Procedure . . . 64
D.6 Aggregate Datatypes . . . 64
D.6.1 Record . . . 64
D.6.2 Set . . . 64
D.6.3 Bag . . . 65
D.6.4 Sequence . . . 65
D.6.5 Array . . . 65
D.6.6 Table . . . 66
D.7 Derived Datatypes and Generators . . . 66
D.7.1 Naturalnumber . . . 66
D.7.2 Modulo . . . 66
D.7.3 Bit . . . 66
D.7.4 Bitstring . . . 66
D.7.5 Characterstring . . . 67
D.7.6 Timeinterval . . . 67
D.7.7 Octet . . . 67
D.7.8 Octetstring . . . 67
D.7.9 Private . . . 68
D.8 Other RPC Datatypes . . . 68
D.9 "Dependent Values" . . . 68
D.10 Cross References . . . 68

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organizations to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13886 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

Annexes A to D of this International Standard are for information only.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Introduction

The purpose of this International Standard is to provide a common model for language standards for the concept of procedure calling. It is an enabling standard to aid in the development of language-independent tools and services, common procedure libraries and mixed language programming. In mixed language applications, server procedures would execute on language processors operating in server mode, and the procedures would be called from language processors operating in client mode. Note that the languages need not be different, and if the processors are the same the model collapses into conventional single processor programming.

Most programming languages include the concepts of procedures and their invocation. The main variance between the methods used in various programming languages lies in the ways parameters are passed between the client and server procedures. Procedure calling is a simple concept at the functional level, but the interaction of procedure calling with datatyping and program structure along with the many variations on procedure calling and restrictions on calling that are applied by various programming languages transforms the seemingly simple concept of procedure calling into a more complex feature of programming languages.

The need for a standard model for procedure calling is evident from the multitude of variants of procedure calling in the standardized languages. The existence of this International Standard for Language-Independent Procedure Calling (LIPC) does not require that all programming languages should adopt this model as their sole means of procedure calling. The nominal requirement is for programming languages to provide a mapping to LIPC from their native procedure calling mechanism, and to be able to accept calls from other programming languages who have defined a mapping to this International Standard.

This International Standard is a specification of a common model for procedure calling. It is not intended to be a specification of how an implementation of the LIPC is to be provided. Also, it is important to note that it does not address the question of how the procedure call initiated by the client mode processor is communicated to the server mode processor, or how the results are returned. The model defined in this International Standard is intended for use by languages so that they may provide standard mappings from their native procedure model. This International Standard depends on the International Standard for Language-Independent Datatypes, ISO/IEC 11404, for the definition of the datatypes that are to be supported in the model for LIPC that it provides.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Information technology – Language-Independent Procedure Calling (LIPC)

1 Scope

This International Standard specifies a model for procedure calls, and a reference syntax for mapping to and from the model. This syntax is referred to as the Interface Definition Notation. The model defined in this International Standard includes such features as procedure invocation, parameter passing, completion status, and environmental issues relating to non-local references and state.

This International Standard does not specify:

- the method by which the procedure call initiated by the client mode processor is communicated to the server mode language processor;
- the minimum requirements of a data processing system that is capable of supporting an implementation of a language processor to support LIPC;
- the mechanism by which programs written to support LIPC are transformed for use by a data processing system;
- the representation of a parameter.

NOTE – Originally it was the intention to align the definitions and concepts of this International Standard with those of the RPC standard (ISO/IEC 11578). Unfortunately, in a late stage of the development process of the RPC standard it was decided to use for that standard a completely different approach. Hence the intended alignment did not materialize.

Annex D gives an overview of the differences between the concepts as defined by this International Standard and the RPC standard.

2 Normative References

The following standards contain provisions which, through reference in this text, constitute provisions of this International Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of current valid International Standards.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

ISO 2375:1985, *Data processing – Procedure for registration of escape sequences.*

ISO/IEC 10646-1:1993, *Information technology – Universal Multiple-Octet Coded Character Set (UCS) – Part 1: Architecture and Basic Multilingual Plane.*

ISO/IEC 11404:1996, *Information technology – Programming languages, their environments and system software interfaces – Language-independent datatypes.*

ISO/IEC 8824-1:1995, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*

ISO/IEC 8825-1:1995, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).*

3 Definitions and Abbreviations

3.1 Definitions

For the purposes of this International Standard, the following definitions apply.

3.1.1 actual parameter: A value that is bound to a formal parameter during the execution of a procedure.

3.1.2 association: Any mapping from a set of symbols to values.

3.1.3 box: A model of a variable or container that holds a value of a particular type.

3.1.4 client interface binding: The possession by the client procedure of an interface reference.

3.1.5 client procedure: A sequence of instructions which invokes another procedure.

3.1.6 complete procedure closure: A procedure closure, all of whose global symbols are mapped.

3.1.7 configuration: Host and target computers, any operating system(s) and software used to operate a processor.

3.1.8 execution sequence: A succession of global states s_1, s_2, ... where each state beyond the first is derived from the preceding one by a single create operation or a single write operation.

3.1.9 formal parameter: The name symbol of a parameter used in the definition of a procedure to which a value will be bound during execution.

3.1.10 global state: The set of all existing boxes and their currently assigned values.

3.1.11 global symbol: Symbol used to refer to values that are permanently associated with a procedure.

3.1.12 implementation defined: An implementation defined feature is a feature that is left implementation dependent by this International Standard, but any implementation claiming conformity to this International Standard shall explicitly specify how this feature is provided.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

3.1.13 implementation dependent: An implementation dependent feature is a feature which shall be provided by an implementation claiming conformity to this International Standard, but the implementation need not to specify how the feature is provided.

3.1.14 input parameter: A formal parameter with an attribute indicating that the corresponding actual parameter is to be made available to the server procedure on entry from the client procedure.

3.1.15 input/output parameter: A formal parameter with an attribute indicating that the corresponding actual parameters are made available to the server procedure on entry from the client procedure and to the client procedure on return from the server procedure.

3.1.16 interface closure: A collection of names and a collection of procedure closures, with a mapping between them.

3.1.17 interface execution context: The union of the procedure execution contexts for a given interface closure.

3.1.18 interface reference: An identifier that denotes a particular interface instance.

3.1.19 interface type: A collection of names and a collection of procedure types, with a mapping between them.

3.1.20 interface type identifier: An identifier that denotes an interface type.

3.1.21 invocation association: The invocation association of a procedure closure <Image, Association> applied to a set of actual parameter values is the association of the closure augmented by a mapping of all local symbols to values and all formal parameter symbols to the corresponding actual parameter values. Thus it is a binding to values of all symbols in the procedure image for the duration of the invocation.

3.1.22 invocation context: For a particular procedure call, the instance of the objects referenced by the procedure, where the lifetime of the objects is bounded by the lifetime of the call.

3.1.23 marshalling: A process of collecting actual parameters, possibly converting them, and assembling them for transfer.

3.1.24 output parameter: A formal parameter with an attribute indicating that the corresponding actual parameter is to be made available to the client procedure on return from the server procedure.

3.1.25 parameter: A parameter is used to communicate a value from a client to a server procedure. The value supplied by the client is the actual parameter, the formal parameter is used to identify the received value in the server procedure.

3.1.26 partial procedure closure: A procedure closure, some of whose global symbols are not mapped. Procedure closures may be complete, with all global symbols mapped, or partial with one or more global symbols not mapped.

3.1.27 procedure: The procedure value.

3.1.28 procedure call: The act of invoking a procedure.

3.1.29 procedure closure: A pair <procedure image, association> where the association defines the mapping for the image's global symbols and no others.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE – Procedure closures are the values of procedure type referred to in ISO/IEC 11404 - Language-Independent Datatypes.

3.1.30 procedure execution context: For a particular procedure, an instance of the objects satisfying the external references necessary to allow the procedure to operate, where these objects have a lifetime longer than a single call of that procedure.

3.1.31 procedure image: A representation of a value of a particular procedure type, which embodies a particular sequence of instructions to be performed when the procedure is called.

3.1.32 procedure invocation: The object which represents the triple: procedure image, execution context, and invocation context.

3.1.33 procedure name: The name of a procedure within an interface type definition.

3.1.34 procedure return: The act of return from the server procedure with a specific termination.

3.1.35 procedure type: The family of datatypes each of whose members is a collection of operations on values of other datatypes. Note, this is a different definition from procedure value.

3.1.36 procedure value: A closed sequence of instructions that is entered from, and returns control to, an external source.

3.1.37 processor: A compiler or interpreter working in combination with a configuration.

3.1.38 server procedure: The procedure which is invoked by a procedure call.

3.1.39 symbol: A program entity used to refer to a value.

3.1.40 termination: A predefined status related to the completion of a procedure call.

3.1.41 unmarshalling: The process of disassembling the transferred parameters, possibly converting them, for use by the server procedure on invocation or by the client procedure upon procedure return.

3.1.42 value: The set Value contains all the values that might arise in a program execution.

3.2 Abbreviations

3.2.1 ASN.1: Abstract Syntax Notation - One

3.2.2 IDN: Interface Definition Notation

3.2.3 LID: Language-Independent Datatypes, as defined in ISO/IEC 11404:1995.

3.2.4 LIPC: Language-Independent Procedure Calling

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

4 Conformance

A language processor may conform to this International Standard by mapping its native procedure calling mechanism to the LIPC model that this International Standard defines.

NOTE – The term "language processor" used in this clause may be extended to include anything which processes information and contains a procedure calling mechanism.

4.1 Modes of conformance

A language processor claiming conformance to this International Standard shall conform in either or both of the following ways.

4.1.1 Client mode conformance

In order to conform in client mode, a language processor shall allow programs written in its language to call procedures written in another language and supported by another processor, using the language-independent procedure calling (LIPC) as provided by clauses 5, 6 and 7 of this International Standard. In this case it is said to conform in (and be able of operating in) client mode. As part of this, the language processor shall define a mapping from its own procedure calling model to the LIPC model.

NOTE – If a program using the LIPC facility is to be portable between processors which conform in client mode, the program and processors will also need to conform to the relevant language standard and the relevant standards binding for that language to the LIPC and LID standards.

4.1.2 Server mode conformance

In order to conform in server mode, a language processor shall allow programs written in another language to call procedures written in its language (i.e. it will accept and execute procedure calls generated by another processor which is executing in a program that is written in that other language and which is operating in client mode, and return control to that client processor upon completion), using the language-independent procedure calling (LIPC) as provided by clauses 5, 6 and 7 of this International Standard. In this case it is said to conform in (and be able of operating in) server mode. As part of this, the language processor shall define a mapping from the LIPC model to its own procedure calling model.

NOTES

1 It is also possible in principle for a client processor to use the model for procedure calls defined in this International Standard to call procedures in the same language; executing on a server processor in the same language, and if the processor conforms in both client and server mode, it is even possible for it to serve itself using this model.

2 If a procedure is to be portable between processors which conform in server mode and the procedure is still to be called by client processors and programs, the procedure, and the processors, will also need to conform to the relevant language standard and the relevant standards binding for that language to the LIPC and LID standards.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5 A model of procedure calling: informal description

5.1 Model overview

A procedure is defined to be a closed sequence of instructions that is entered from, and returns control to, an external source.

The general structure of a procedure call can be described as a single thread of execution in a particular program where the flow of control is passed from one procedure to another. The originator of the call is known as the client procedure and the procedure being called is referred to as the server procedure.

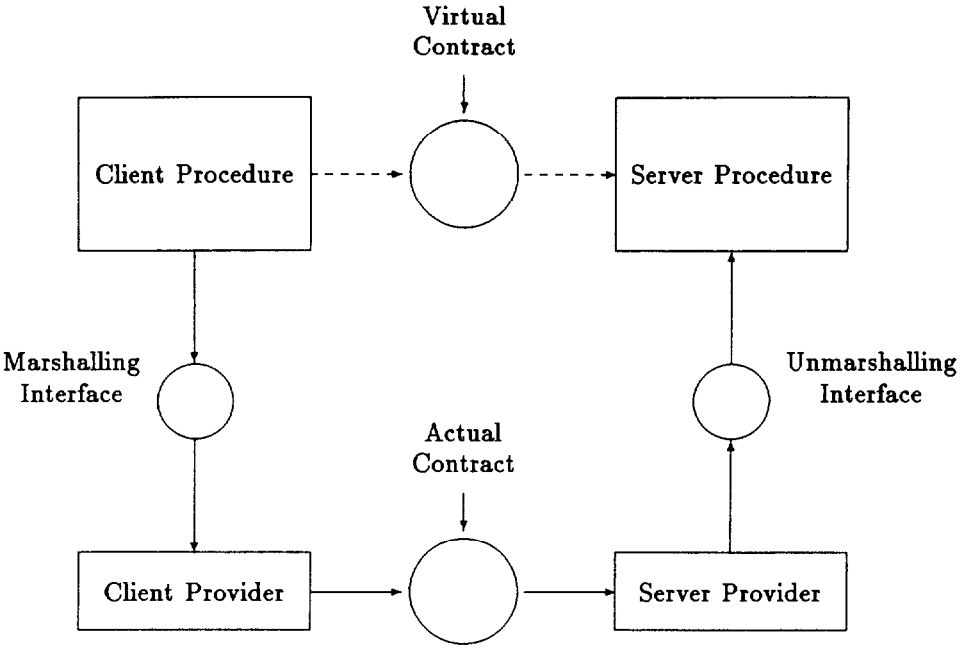
NOTE 1 – It is possible for a server procedure to also be a client procedure if it makes a call to another procedure in order to complete its desired function.

Procedures have the ability to exchange data between the client and server via the use of parameters (see 5.2). In addition, client and server procedures may also share data through the use of global data (see 5.2.2). In order for the parameters specified by the client procedure to be interpreted correctly, the parameters are required to be marshalled (see 5.2.3) to a base form for transmission that is shared by both the client and the server procedure. After the data has been transmitted, the server procedure must then unmarshall (see 5.2.3) the data from the base form into datatypes that are defined in the server language or in the language binding to ISO/IEC 11404 - Language-Independent Datatypes for that particular language.

NOTE 2 – An example of the process of marshalling and unmarshalling of parameters would be if a Pascal client procedure made a call to a Fortran server procedure passing a single character parameter by value. The Pascal "char" datatype would map to a LID character. In order to have the LID character be transmitted to the server procedure, the LID character is marshalled to an appropriate ASN.1 value, for example, which is a form that would be understood by both the client and server procedures. The ASN.1 value would then be transmitted to the server and upon receipt it is unmarshalled into a LID character, which in turn maps to a "character*1" in Fortran.

The following diagram outlines the basic components of the language-independent call model:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale



Language-Independent Procedure Call Model

This model illustrates how the client and server procedures communicate when their implementations conform to this International Standard. The virtual contract between the client procedure and server procedure is defined by the Interface Definition Notation contained within this International Standard. Upon the instantiation of a call, the marshalling interface marshalls the parameters and passes this information on to the client LIPC provider. The client LIPC provider is connected to the server LIPC provider via the actual contract which is the transmissible form (e.g., ASN.1). The server LIPC provider then unmarshalls the data, via the unmarshalling interface, into a form that is compatible with the server procedure. Upon return, the process is reversed with the unmarshalling interface now being the marshalling interface and the marshalling interface now being the unmarshalling interface.

5.2 Parameter passing

Any datatype defined in ISO/IEC 11404 - Language-Independent Datatypes can be the datatype of a formal parameter of a language-independent procedure call. This International Standard defines parameter passing solely on the passing of values. Therefore an actual parameter is any value of the datatype required by the call. The parameter passing model defined in this International Standard is a strongly typed model.

NOTE 1 – Weak typing can be accomplished by relaxing association rules and adding implicit datatype conversions in the language bindings to this International Standard.

The following notes relate the common parameter passing mechanisms that are found in existing languages to the four defined parameter passing schemes that are defined in this International Standard.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTES

2 Call by Value (In parameters): This is the simplest of all common parameter passing mechanisms and appears directly in LIPC as Call by Value Sent on Initiation (see 5.2.1.1). The virtual contract is fulfilled by the client evaluating the actual parameter and sending the value to the server procedure, and the server procedure accepting it. No further action is required of the client procedure. The server procedure does what it likes with the received value, but can make no further demands on the client with respect to the actual parameter that generated the value.

3 Call by Value Return (Out parameters): This common parameter passing mechanism is also directly supported in LIPC by Call by Value Returned as Specified. The virtual contract for this mechanism involves the concept of passing a parameter only as a means of receiving a value. If in a specific language binding, a parameter is passed at the language processor level, what is passed is an implicit pointer to a value of the datatype concerned, which the server procedure contracts to set. The server procedure cannot access the value of the datatype prior to the call. Some languages, in their datatyping model, explicitly distinguish between the datatypes of values held by variables and those of the variables themselves. For example, some languages have an explicit dereference (i.e., obtain the value of). For languages without such a model, the LIPC model allows that distinction to be made at the language binding service contract level without disturbing the virtual contract model.

4 Call by Value Sent and Return (In-out parameters): This common parameter passing mechanism is an in/out mechanism where the actual parameter can be evaluated to a destination for Call by Value Returned on Termination (see 5.2.1.3). However, in the LIPC model it is regarded as a parameter with both that property and that of Call by Value Sent on Initiation (see 5.2.1.1). Equivalently, it can be expanded into two implicit parameters, one of each kind.

The actual parameter corresponding to a formal parameter of a given datatype "t" must be capable, on evaluation, of yielding a destination for such a value (i.e., an implicit or explicit pointer to a value of datatype "t"). For the "in" part of the in/out specification, the current value held in that destination on initiation of the call is retrieved by the client and relayed to the server procedure. The destination itself is also recorded. In the virtual contract the client receives the returned value, the "out" part of the in/out specification, from the server procedure and sends it to that destination.

Where the language binding or service contract passes the destination itself to the server procedure as part of the copy-in/copy-out, the server procedure must contract to retrieve the "in" value immediately on transfer and then to send the returned "out" value to the destination on completion of the call. While the call is in progress, the client explicitly or implicitly marks the destination as 'read once only, write once only' as far as the server procedure is concerned and any attempt by the server procedure to violate that condition is an error.

5 Call by Reference: In this case a formal parameter of datatype "t" is interpreted as an implicit 'pointer to "t"' and the actual parameter must evaluate to such a pointer accordingly. This pointer to "t" is then passed by value as an "in" parameter.

The pointer is not passed as an in/out parameter since this would cause an extra level of indirect addressing.

The virtual contract is that the client provides an access path to the destination. The destination is fixed, but the access path can be used by the server procedure both reading and writing of values of datatype "t". In the close-coupled case the service contract may well involve passing the actual destination with the client needing to take no further action until the call is complete. In a loosely-coupled service environment the service contract will involve client action during the call, responding to requests by the server for a value of datatype "t" to be read or written. In effect this would be reciprocal calls with the "in" and "out" directions reversed.

These reciprocal calls implied by Call by Reference in a loosely-coupled environment represent a potentially significant overhead, which may result in Call by Reference not being supported

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

in such services.

5.2.1 Methods of parameter passing

There are four basic kinds of parameter passing defined in this International Standard:

1. Call by Value Sent on Initiation
2. Call by Value Sent on Request
3. Call by Value Returned on Termination
4. Call by Value Returned when Available

5.2.1.1 Call by Value Sent on Initiation

This is the simplest form of parameter passing. The formal parameter of the server procedure receives a value of the datatype concerned. The virtual contract is that the client evaluates the actual parameter and supplies the resulting value at the time of transfer of control. The server procedure accepts this value and no further interaction takes place with respect to this parameter.

NOTE – This type of parameter passing is commonly known as Call by Value.

5.2.1.2 Call by Value Sent on Request

The virtual contract for this type of parameter passing is that the client undertakes to evaluate the actual parameter and supply the resulting value, but only upon receipt of a request to do so from the server procedure. The evaluation and passing of the actual parameter takes place if and only if the server procedure requests it. This can be done at the beginning of the call, or while the call is in progress.

The essential difference from Call by Value Sent on Initiation is that in some cases the value sent will be different.

NOTES

1 While this mechanism is not common to programming languages as an explicit standards requirement, it is an optimization mechanism for programming language implementations.

2 An example of the use of Call by Value Sent on Request is when a client wishes the server procedure to record a time, and wishes that to be done at a specific point during the execution of the call, rather than at the initiation of the call.

3 The use by the server of a parameter of the Call by Value Sent on Request type can be regarded as a call of an implicit procedure parameter where the server procedure does the evaluation one time. Any further reference in the server procedure to the formal parameter simply uses that same value. The server procedure does not issue a further request for a value.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.2.1.3 Call by Value Returned on Termination

In this type of parameter passing, the virtual contract is that at the completion of the call, the server procedure will supply a value of the datatype of the formal parameter and the client will accept it and send the returned value to the appropriate destination.

NOTE 1 – This type of parameter passing is better known as Call by Value Return and is essentially the "out" equivalent of Call by Value Sent on Initiation.

Conceptually the client and not the server procedure sends the returned value to the destination, because the client language or mapping determines the interpretation of the destination and the process of return.

NOTES

2 In a closely coupled environment where providing the actual destination (perhaps even the hardware address) to the server procedure is a trivial task, there is no reason why the actual service contract at the implementation level should not include providing the actual destination to the server procedure, which then sends its returned value directly there. This is an additional service level function that the server procedure contracts to perform for the client procedure, which does not affect the logical division of responsibility at the virtual contract level.

3 This kind of parameter passing also accommodates the return of a value for the procedure as a whole, in the case of function procedures. Parameter passing utilizing Call by Value Returned on Termination accommodates function procedures through the use of an additional anonymous parameter.

5.2.1.4 Call by Value Returned when Available

In this type of parameter passing, the server procedure returns the parameter value at any time after the returned value is available. It could be returned while the call is still in progress, at the completion of the call, or some time later. What time is chosen is determined by the binding of the LIPC based service and is not defined by this International Standard. All the LIPC model requires is that this possibility can be accommodated. The virtual contract is that whenever the server procedure returns the value, the client will accept it and send the returned value to the appropriate destination.

NOTE – In this type of parameter passing, the possibility that the returned value will be returned more than once is not excluded.

5.2.2 Global data

The term *global data* is used for data defined in a shared execution context that can be referenced by another procedure executing in a different invocation context within the same execution context. Conceptually, global data requires the marshalling/unmarshalling of global data into individual information units. Implementations conforming to this International Standard may support an implementation-defined mechanism for the sharing of global data and may support partitioning of global data. Partitioning of data refers to the ability to insulate data from a procedure. It is

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

recommended that implementations support global data via implicit parameters that are passed on the call, but this may not be the only valid mechanism where the marshalling/unmarshalling operations are known to be trivial.

NOTES

1 In the IDN, global data is represented as an explicit parameter to the procedure. In a particular language mapping, these explicit parameters can be provided to the procedure using such mechanisms as external variables and as such are implicit parameters.

2 Global data should be available to the server by the time it is needed (i.e., before invocation, at invocation, before use is required, or at the time access is required).

3 The mechanism by which objects in the invocation context are associated to the global objects may be defined by the language, language mapping, or left to the implementation.

5.2.3 Parameter Marshalling / Unmarshalling

Data which is communicated between the client and server procedure needs to be assembled in a transmissible form. This transmissible form will allow the client and server procedures to encode their LID mapped data into a form that is suitable for both language-independent calling on the same system and remote procedure calls. The specification of this transmissible form is outside the scope of this International Standard.

NOTE – The Abstract Syntax Notation - One is a suitable specification of a transmissible form.

The marshalling of data refers to what the client procedure must do in order to transform its data into a form for transmission to the server procedure. Unmarshalling of data refers to what the server procedure must do in order to take the data passed by the client procedure and transform this into data suitable for the language of the server procedure. Marshalling is not limited to calling a procedure. Upon return, the server procedure must marshall any returned data into the form shared by the two procedures. Unmarshalling of data is not limited to the server procedure, since the client procedure must be able to unmarshall any data that is returned by the server procedure.

Since marshalling and unmarshalling of data for procedure calls is often complex and degrades performance, an implementation may want to perform optimization of this process wherever possible. Optimizations will likely be available when the client and server systems are homogeneous and the languages involved in the procedure call have the same data representation.

5.2.4 Pointer Parameters

A Call by Value Sent on Initiation of a pointer allows access to the entity pointed to. The pointer value itself cannot be changed by the server procedure in order for the pointer to refer to something else after the call.

NOTE – For example, if the value sent is a pointer to a record, after the call the pointer still points to the same record even though the values in the fields of the record may have changed.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

If changing what the pointer refers to is needed, then another level of indirect referencing has to be invoked, either directly (as with call by reference) or indirectly (as with call by value-return). An access path via pointer parameters implies access to all lower levels, including the primitive datatype values referenced by the lowest level pointers.

5.2.5 Private types

A private type is a datatype that is protected from modification within the server procedure regardless of the attributes on a parameter being passed as a private type. No operations shall be permitted on a parameter of a private type. A private type is declared by including the `restricted` keyword prior to the LID datatype in the IDN.

> NOTE – A private type can be considered as an octet stream that can have no operations performed on it.

5.3 Execution-time Control

5.3.1 Terminations

An implementation conforming to this International Standard shall provide a method for raising and handling terminations that occur during the initialization, execution, or completion of a procedure call. Raising a termination does not necessarily imply that the server procedure should be terminated immediately, however terminations that are raised must not be ignored by the implementation. Some examples of possible terminations include:

- normal termination of a procedure invocation returning the output parameters, input/output parameters, and result (if any)
- abnormal termination, in which the procedure itself detects an error or other unusual condition
- external cancellation, in which some other entity determines that the procedure should terminate
- hardware or software detected events which may or may not be critical to the proper execution of the application
- asynchronous events or notification
- type or value mismatches in parameter passing or return
- failure of the underlying invocation service itself.

5.3.1.1 Normal termination

A procedure completing normally raises a termination signifying a normal return. A procedure may report additional terminations; e.g., at return from a synchronous procedure call, the procedure may return two or more terminations; however, the first of these terminations must specify whether termination is normal, abnormal, or via a cancel. If the procedure call is asynchronous, the procedure may return an additional termination code before, during, or after termination.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

5.3.1.2 Abnormal Termination

A procedure completing abnormally raises a termination as a result of some condition other than an external cancel command. The usual reason a procedure abnormally terminates is that the procedure encounters some condition that makes it impossible to continue or impossible to complete successfully the function(s) requested by the client procedure. The client procedure is notified by the implementation defined termination raising mechanism. Abnormal terminations can be divided into two cases:

- a procedure detects an abnormal termination as part of its logic and executes an explicit abort procedure as a result
- an abnormal termination occurs during execution of the procedure, causing a fault at some level lower than that of the procedure logic; the fault causes control to go to some generic fault-handling routine within the procedure that terminates the procedure as in the previous case.

A special case of abnormal termination of a procedure is the case where one or more of the actual parameter values in the procedure call are incorrect, e.g., a value is of the wrong datatype for a given parameter or of the right datatype but outside the required range. It is possible to distinguish here between parameter values that violate the advertised requirements of the procedure interface as specified in the IDN and values (or combination of values) that violate application specific constraints that cannot be specified in the IDN formalism and hence must be checked explicitly by the procedure itself. However, from the point of view of the client procedure, the only difference between the two cases is that in the first case, the error specified is one of a predefined set specified in this International Standard (see 5.3.1.4). In the second case, it is an application-specific condition code specified in some other, perhaps application-specific, standard.

5.3.1.3 External Cancellation

A procedure terminates by external cancellation if a command is issued from outside the procedure which causes the procedure to terminate, or be terminated, in an orderly way. In the case of an asynchronous call, the cancellation may come from the client procedure. Whether the call is synchronous or asynchronous, the command to cancel a procedure may come from an outside source, i.e., outside the LIPC model. The two cases are indistinguishable to the server procedure. In both cases, the client procedure receives a notification via an implementation defined termination raising mechanism.

5.3.1.4 Predefined conditions

As a minimum, implementations conforming to this International Standard should report the following terminations during a procedure call:

- server procedure unavailable, call not executed
- client or server procedure does not have defined mapping to IDN

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- value out of range for parameter datatype
- cancellation of call
- insufficient resources available to complete call
- normal completion of call

5.4 Execution Control

5.4.1 Synchronous and Asynchronous Calling

The issue of whether of not a call executes synchronously or asynchronously is outside the scope of this International Standard. The LIPC model does not prohibit either synchronous or asynchronous calls. An implementation can choose whether or not to limit the number of threads of execution in any particular call environment.

5.4.2 Recursion

The LIPC model does not prohibit recursion. How an implementation implements recursive procedure calling is outside the scope of this International Standard.

NOTE – Implementors should be aware that optimization considerations for LIPC calls need to take recursion into account.

6 A model of procedure calling: formal description

This clause provides a model of procedures, variables, name bindings, execution environments, and invocation. A series of new datatypes are introduced. Some of these directly correspond to programming concepts (like variables), and some are used merely to support further definitions.

6.1 Value

The set Value contains all the values that might arise in a program execution. Value contains all the values definable using the datatypes, type generators, and definitional mechanisms of ISO/IEC 11404 - Language-Independent Datatypes. Value will also contain boxes and procedure closures (see 6.6).

6.2 Boxes and global state

A box is a generic term for a container that holds a value of a particular datatype, for example what, in some contexts, would be called a 'variable'. Boxes exist and are manipulated at execution-time. They may be named by identifiers in some program text, but they are distinct from any such syntactic notion. Boxes do not imply any particular implementation mechanism such as storage. There are three operations defined on boxes:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
create:              → Value

write:   Box * Value →

read:    Box         → Value
```

The above three lines are called *signatures*. Each signature lists the name of an operation, the types of the inputs (if any) of that operation, and the types of its outputs (if any). An * (the cartesian product operator) separates input (or output) types.

The operation Create brings a new box into existence. The operation Write associates a new value with a given box. The operation Read returns the last value written to a given box. If read is applied to a box that has never been written, the value returned is unspecified.

The global state is the set of all existing boxes and their currently assigned values. It is the unique characteristic of boxes that their operations involve global state: the operation Read accesses the global state; the operation Create and the operation Write produce a new global state.

NOTE 1 – The global state exists as a modelling concept only. No individual program, executing on a particular machine, can access all parts of the global state. It is a characteristic of distributed systems that each part of the system can only access a few 'local' boxes, and must ask other 'remote' parts of the system to read or write 'remote' boxes.

Boxes also imply a notion of time, modelled as a point in an execution sequence. An execution sequence is a series of global states s_1, s_2, ... where each state beyond the first is derived from the preceding one by a single create operation or a single write operation.

NOTE 2 – In cases of concurrent processing, the series of global states making up the execution sequence cannot necessarily be determined by examining the program text and may vary from execution to execution.

6.3 Symbol

A symbol is a reference in a program text to a value of a particular datatype (including boxes and procedure closures). These referenced values are the values that the procedure can access directly during execution. The symbols of a particular procedure fall into three disjoint categories:

- Global symbols are used to refer to values that are permanently associated with the procedure (e.g., other procedures, non-local variables, or 'own' variables).
- Local symbols are used to refer to values that exist only for the duration of a single invocation (e.g., the local 'stack frame' variables).
- Parameter symbols are the formal parameters used to refer to values that are the actual parameters for a particular invocation.

NOTES

1 Local symbols and parameter symbols of one procedure may be global symbols of another procedure (e.g., nested procedures).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

2 How references to values in a program text in a particular language are expressed is defined by the rules of the language, including its scoping rules. For example, the means of reference may be an identifier and the same identifier may relate to two different references in different program contexts (because of scoping rules). The identifier would thus correspond to two different "symbols" in the sense of this subclause.

3 By binding global symbols to boxes, these global symbols can (indirectly) refer to values created at arbitrary times, and be associated with the given procedure for arbitrary periods. Thus the phrase "permanently associated" above is not a substantive restriction to what can be modelled.

6.4 Procedure image

A procedure image is the abstraction of a procedure text. Implicit in a procedure image is the procedure-type, the global, local, and parameter symbols used within the procedure text, and the algorithm to be executed by the language processor. There are four operations defined on procedure images:

```
gsyms:  Image → Sequence(Symbol)

lsyms:  Image → Sequence(Symbol)

psyms:  Image → Sequence(Symbol)

spec:   Image → Procedure_Type
```

NOTE 1 – The ordering within the sequence produced by *gsyms* and *lsyms* is seldom relevant, however the ordering within the sequence produced by *psyms* is important (see 6.11).

Gsyms returns the global symbols of the image. *Lsyms*, *psyms*, and *spec* return (respectively) the local symbols, parameter symbols, and the procedure type.

NOTE 2 – Procedure images are created by the language processor. How a procedure image is created is outside the scope of this International Standard.

6.5 Association

An association is any mapping from a set of symbols to values.

```
A: Symbol → Value
```

Associations are typically partial, being defined only on the symbols used by a particular procedure image. Let x be a symbol, y a value, and A and B be associations.

```
[x → y]   denotes the association that maps the symbol x
                  to the value y and maps no other symbols
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
A + B       denotes an association that satisfies

            (A+B)(x) = B(x)  if B is defined on x

                     = A(x)  otherwise

domain (A) denotes the set of symbols x for which A(x) is defined

range (A)  denotes the set of values { A(x) | x is in domain(A) }
```

6.6 Procedure closures

A procedure closure is a pair <I,A> where I is a procedure image and A is an association mapping the global symbols of I, and no others. In particular, the local and parameter symbols have no mappings. Procedure closures are the values of procedure type referred to in ISO/IEC 11404 - Language-Independent Datatypes.

A *complete* procedure closure is a procedure closure for which all the global symbols of the image are mapped.

A *partial* procedure closure is a procedure closure for which at least one of the global symbols is not mapped.

NOTES

1 An example of a partial procedure closure is the value of a procedure A nested within a procedure B before procedure B is invoked. This is partial because references from A to B's local variables cannot be mapped until the invocation of B.

2 Procedure closures are typically constructed as part of compilation, or during execution, according to the rules of the particular programming language involved.

6.7 Boxes, pointers, values, and datatypes

A pointer datatype, as defined in ISO/IEC 11404 - Language-Independent Datatypes, is a datatype whose values are references to other values; in particular, a value of datatype pointer-to-D, where D is a datatype, is a reference to a value of datatype D.

In the LIPC model, the datatype of a box is a pointer datatype as defined in ISO/IEC 11404 - Language-Independent Datatypes Every box has a value which is a reference to some other value. If a box is used (e.g., in a LIPC mapping) to model the concept of "a variable of datatype D", which some languages have, then it "holds" a value of datatype D (see 6.2) and the box is a value of datatype pointer-to-D.

NOTE 1 – An entity called "a variable of datatype D" cannot literally be a value of datatype D because such values cannot vary, any more than an "integer array" can literally be a value of datatype integer (since those are single values only).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

In ISO/IEC 11404 - Language-Independent Datatypes, "dereference" is defined as a characterizing operation of all pointer datatypes. When this operation is applied to a value P of datatype pointer-to-D, the result is the value V of datatype D that P references. In the LIPC model, the corresponding operation on a box is Read. The Create and Write operations for boxes (see 6.2) are not characterizing operations defined in ISO/IEC 11404 - Language-Independent Datatypes, but correspond respectively to situations where new objects of pointer datatype can be created, and when the value V of datatype D referenced by a particular pointer-to-D value P is replaced by a new value. Both of these operations are needed for boxes in the LIPC model though neither are necessarily required for all objects of pointer datatype in all circumstances.

NOTES

2 The operation Write on a box corresponds to the concept in many languages of "assigning a value". Changing the value of datatype D referenced by a box does not change the box itself, only its contents, just as assigning a new value to a variable X in a language does not change X itself, which is still the same variable with the same name.

3 The concept of "pointer variables", sometimes referred to as "indirect addressing", exists in some languages. ISO/IEC 11404 - Language-Independent Datatypes defines the datatype of such an object as pointer-to-pointer-to-D. It references a value of datatype pointer-to-D, (i.e., an object of another pointer datatype). That object references a value of datatype D. In the LIPC model this corresponds to a box which holds a reference to another box, which in turn holds a value of datatype D.

In this way, the LIPC model supports parameter passing of pointer datatypes, which some languages support directly or indirectly, and both the LID and LIPC standards support indirect addressing of any required depth.

4 In some discussions of programming languages the concept "instances of values" is used. In LIPC terms an instance of a value can be thought of as a value being held in a box, which allows modelling of situations where multiple instances of the same value exist simultaneously. It is possible in the LIPC model for more than one entity to have access to the same box (e.g., it has been passed as an actual parameter whose formal parameter is of a pointer datatype). The entities than have access to the same instance of the value; furthermore, if the box is modified (i.e., the value it holds is changed through use of a Write operation), then this modification is visible to both entities and hence may affect subsequent behavior in either or both. In environments in which such multiple accesses cannot be supported directly, some implementation defined mechanism must be provided to simulate it, for example by creating duplicate boxes, and providing means of ensuring that any change in one is automatically applied to the other, either immediately or at least before any event occurs which uses the value held in the box.

A procedure datatype, as defined in ISO/IEC 11404 - Language-Independent Datatypes, is in general a composite (though not an aggregate) datatype which incorporates within its specification the datatype of all of its parameters (including any notional parameters used to return results to "function" procedures). In the LIPC model, a procedure closure is an entity whose datatype is some LID procedure datatype, but also encompasses the concept of Global State.

NOTE 5 – By this means, matching of procedure datatypes in the LIPC model automatically ensures matching of the number and datatypes of parameters.

6.8 Interface closure

An interface closure is a collection of names and a collection of procedure closures, with a mapping between them. This is modelled as an association that maps the set of names to procedure closures.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE – For example, if Sue, Mary, and Sam are procedure names (symbols), and X, Y, and Z are procedure closures, then

I = [Sue → X] + [Mary → Y] + [Sam → Z]

is an interface closure. domain(I) = {Sue, Mary, Sam} (see 6.5). Thus, domain(I) is the set of procedure names in the interface closure I. The procedure closure in I named by Mary is denoted I(Mary).

6.9 Interface type

An interface type is a collection of names and a collection of procedure types, with a mapping between them. This is modelled as an association that maps a set of procedure types to names.

NOTES

1 An interface type is not a datatype.

2 For example, if Sue, Mary, and Sam are procedure names, if X, Y, and Z are procedure closures of the corresponding procedure types XT, YT, and ZT respectively, then

IT = [Sue → XT] + [Mary → YT] + [Sam → ZT]

is the interface type corresponding to the interface closure I introduced in the preceding sub-clause.

6.10 Specifications

The LIPC model defines the operation *spec* on procedure images to return the procedure type of the image. Thus,

spec: Image → Procedure_Type

spec is generalized to procedure closures by

spec: Procedure_Closure → Procedure_Type

spec (<image,assoc>) = spec (image)

and spec is further generalized to interface closures by the following where P is a procedure_type and I is an interface_closure:

spec: Interface_Closure → Interface_Type

For I = [$name_1$ → P_1] + ... + [$name_n$ → P_n],

spec (I) = [$name_1$ → spec(P_1)] + ... + [$name_n$ → spec(P_n)]

The operation spec on interface_closures returns an association between names and procedure types.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.11 Basic procedure invocation

Basic procedure invocation is an operation on complete procedure closures described as follows:

invoke: Procedure_Closure * Sequence(Value) → Status * Sequence(Value)

where Status is the set of termination identifiers (see ISO/IEC 11404:1996, clause 9.3). The first sequence of values represents the input parameters to the invocation. The second sequence of values represents the values resulting from the invocation. The status represents the termination condition, including the "normal" termination.

Applying invoke to the procedure closure <Image,Association> and input values $<V_1,...V_n>$ results in the following actions:

Let $<A_1,...A_n>$ = psyms(Image)
$<L_1,...L_m>$ = lsyms(Image)

For i = 1 to m, do

LB_i = create()

Define invocation association Q by:

Q = Association + $[A_1 \rightarrow V_1]$ + ... + $[A_n \rightarrow V_n]$
+ $[L_1 \rightarrow LB_1]$ + ... + $[L_m \rightarrow LB_m]$

Then

''Execute Image in the context of association Q''

Executing an image in a context is a primitive notion defined by the programming language processor (or standard) for the language in which Image is written. When execution terminates, a value in Status * Sequence(Value) will result, and the association Q is lost.

NOTE – The boxes created in forming Q are no longer accessible unless the programming language permits them to be "returned" in some fashion (see 6.13)

6.12 Type correctness

It is not meaningful to apply the invoke operation to any procedure closure C and any sequence of input values $<V_1,...V_n>$. Invocation is meaningful only if its parameters are type correct.

Let spec(C) be

PROCEDURE (a_1: AT_1, ... a_{an}: AT_{an})
RETURNS (r_1: RT_1, ... r_{rn}: RT_{rn})
SIGNALS (E_1, ... E_{en})

where a_1 through a_{an} are the input parameters (in order), r_1 through r_{rn} are the output parameters (in order), and E_1 through E_{en} are the non-normal terminations.

Invocation of C on $<V_1,...V_n>$ is type correct if

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

n = an (the number of parameters is correct)

and

For i = 1 to n,
V_i is a value of type AT_i (the types of the parameters are correct)

When the invocation of C on $<V_1,...V_n>$ terminates, it produces a result of the form

$< \text{status}, <W_1,...W_m> >$

If the closure C is a member of the procedure type to which it has been mapped, then the following information is known about the above result.

status = ''normal'' or status = E_i for some i in 1..en
If status = ''normal'' then
m = rn, and
W_j is a value of type RT_j (for all j in 1..m)
ElseIf status = E_i, and E_i is declared to have structure
f_1: FT_1, ... f_{fn}: FT_{fn}
then
m = fn, and
W_j is a value of type FT_j (for all j in 1..m)

6.13 Associates

In order to be able to discuss the set of global variables shared by two procedures, or to define pointer (or parameter) aliasing, it is necessary to know when one value is "referenced by" or is "accessible from" another value. X is a *simple associate* of Y if X can be obtained from Y by following pointers or extracting the elements of aggregate values. X is a *generalized associate* of Y if X can be obtained from Y by combinations of the above actions and by invocating procedures.

The next two clauses formalize these two concepts.

6.13.1 Simple Associates

The concept of simple associates is embodied in two functions.

The first such function is Immediate Associates. IAssoc(x) is defined as follows:

IAssoc: Value → Set(Value)

- If x is a value of some non-aggregate type defined in ISO/IEC 11404 - Language-Independent Datatypes, then IAssoc(x) is the empty set.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

- If x is a value of some aggregate type defined in ISO/IEC 11404 - Language-Independent Datatypes, then IAssoc(x) is the set of all elements of the aggregate.
- If x is a box which currently holds a value v,

  `IAssoc(x) = the set consisting of the single value v`

- If x is a procedure closure,

  `IAssoc(x) = the empty set`

The second associates function is Transitive Associates:

`Assoc: Value → Value`

For any value x, Assoc(x) is the smallest set satisfying

x is in Assoc(x)

If y is in Assoc(x), then all elements of IAssoc(y) are in Assoc(x).

NOTE 1 – Intuitively, Assoc(x) consists of all values that can be extracted from x by applying various extraction operations on aggregates and read operations on boxes. Since read depends on the current state, IAssoc and Assoc depend on the current state as well.

When a procedure closure <I,A> is invoked on inputs $<V_1,...V_n>$, it has immediate and direct access to all the values in

$range(A) \bigcup \{V_1,...V_n\}$

and (with some computation) direct access to all the values in

$Z = Assoc\ (\ range(A) \bigcup \{V_1,...V_n\}\ )$

The invocation of <I,A> on $<V_1,...V_n>$ can potentially read or write any box in Z and no others.

NOTES

2 It can also create new boxes.

3 The set Z does not include values that can only be accessed by invoking other procedure closures.

6.13.2 Generalized Associates

The concept of generalized associates includes values that can be obtained with the help of other procedures. Again, two functions are needed.

Generalized Immediate Associates is defined as follows:

`GIAssoc: Value → Value`

If x is a procedure closure <I,A>,

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
GIAssoc(x) = range(A).
```

If x is any other value,

```
GIAssoc(x) = IAssoc(x).
```

Generalized Transitive Associates is defined as:

```
GAssoc:  Value → Value
```

For any value x, GAssoc(x) is the smallest set satisfying

is in GAssoc(x)

If y is in GAssoc(x), then all elements of GIAssoc(y) are in GAssoc(x).

NOTE – Intuitively, GAssoc(x) consists of all values that can be extracted from x by applying various extraction operations on aggregates, read on boxes, and procedure invocation.

When a procedure closure <I,A> is invoked on inputs $<V_1,...V_n>$, let

$GZ = GAssoc\ (\ range(A) \bigcup \{V_1,...V_n\}\ )$

With the help of other procedure closures, this invocation can access any value in GZ. The only elements of the global state that the invocation of <I,A> on $<V_1,...V_n>$ can access or modify are those which are boxes in GZ.

It is assumed that for a procedure closure <I,A> to invoke another procedure closure <J,B>, <J,B> must be one of the following alternatives:

1. in range(A) (the most common case)
2. accessible from an input parameter,
3. J is in the range of the invocation association of <I,A> and B can be constructed from accessible values.

6.14 Execution and Invocation contexts

The execution context of the procedure closure <Image,Association> is the set of all boxes in the Assoc(range(Association)). The invocation context of a particular invocation of the procedure closure <Image,Association> is the set of all boxes in the Assoc(range(Q)) where Q is the invocation association of this invocation of <Image,Association>.

NOTE – Both the execution context and the invocation context can vary over time during the execution of Image.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.15 Parameter translations

When a procedure invocation is required to cross between execution contexts, it may not be possible to pass the parameter and return values directly between these contexts. Consider the following two examples.

In the first example, a program written in programming language L1 calls a procedure written in language L2. If L1 and L2 have different datatypes, this call may require translating L1 input values into their L2 equivalents. On return a reverse translation may be needed.

In the second example, a program calls a procedure written in the same language (thus needing no datatype translation), but in a separate address space. Assume that pointers are implemented in a way that ties them to a specific address space (the usual case). So any pointers in the input values will be tied to the client procedure's address space. These pointers must be uniformly replaced by "equivalent" pointers tied to the procedure's address space. Again, on return a reverse translation may be needed.

Parameter translations can lose information (e.g., when translating between different floating point formats), and can disrupt sharing relationships (e.g., when moving pointers between address spaces). Since these effects are visible to programmers, the LIPC model defines a way of handling them.

Parameter translations are modelled in the following way. Let C be an arbitrary procedure closure, and let TF and TB be procedure closures that do parameter translations for C. Then we will define

```
wrap (TF, C, TB)
```

to be a procedure closure that (when invoked) does the following:

1. invokes TF to translate the input parameters
2. invokes C with the translated parameters, and
3. invokes TB to translate the returned values back again.

The following describes how the wrap function aids in modelling cross execution context calls. Let X1 and X2 be execution contexts.

> NOTE 1 – It does not matter what an execution context is, just that some sort of translation is necessary to call from one to the other.

Let C1 be the procedure closure representing the target procedure in its native context X1. Then,

```
C2 = wrap (TF, C1, TB)
```

is the procedure closure which is actually called in context X2. In many cases, calling C2 will have visibly different effects from calling C1.

A more precise definition of wrap would be:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
wrap:  Procedure_Closure * Procedure_Closure * Procedure_Closure
       → Procedure_Closure
wrap (TF, C, TB) = <IM, [pre→TF] + [main→C] + [post→TB]>
```

For convenience, assume that TF and TB take a single Sequence(Value) input and produce a single Sequence(Value) output. This allows TB in particular to be invoked on output sequences of differing length.

When procedure closure wrap(TF,C,TB) is invoked on input sequence V the image IM causes the following steps to occur:

1. TF is invoked on <V>, producing <E, W>

   (1.1) If E ¬= ''normal'', IM terminates with <E, W>
   (1.2) If E = ''normal'', W is a singleton $<W_1>$

2. C is invoked on W_1, producing <F, X>

3. TB is invoked on <X>, producing <G, Y>

   (3.1) If G ¬= ''normal'', IM terminates with <G, Y>
   (3.2) If G = ''normal'', Y is a singleton $<Y_1>$

4. IM terminates with $<F, Y_1>$

Using procedure closures to do the parameter and result translations allows the full computational power of the model to be used in expressing these translations. TF and TB can communicate with each other via shared boxes, and can access arbitrary other parts of the global state if their association maps are defined accordingly. However in typical usage, TF and TB are expected to be quite simple.

NOTE 2 – TF and TB are the only places where Value (the union of all datatypes) is used in a conceptual context.

Example: model of a remote procedure call from a client address space (CAS) to a server address space (SAS). Let P be a procedure in SAS. Let MC be the client side marshalling code, and UC be the client side unmarshalling code. MS and US are the corresponding codes on the server side. The procedure closure

```
PW = wrap (US, P, MS)
```

represents procedure P as exported to the outside world. PW takes ''wire format'' data as input and returns ''wire format'' data as output. The procedure closure

```
PC = wrap (MC, PW, UC)
```

represents procedure P as imported into CAS. PC's inputs and outputs are appropriate for CAS.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

6.16 Defining Translation Procedures

Translation procedures typically need to take a composite value V and replace only certain portions of V, leaving the rest of V as in the original. An example is replacing all boxes in V with new ones while preserving the sharing structure within V. Expressing this as an algorithm can be somewhat complex. However there are a number of concepts that can help describe the intended result (leaving the algorithmic details to the implementors).

The following definitions are not used in this International Standard, but will help shorten definitions in binding standards.

Let T be some mapping from values to values:

$$T: \text{Value} \rightarrow \text{Value}$$

T is *an identity on* datatype Q if for all values v of datatype Q,

$$T(v) = v$$

Let F be a characterizing operation of datatype Q, and F' be a characterizing operation on datatype Q' with the same number of parameters as F. T *maps* F *to* F' if for all values $v_1,...v_n$ in the input domain of F,

$$T(F(v_1 \ldots v_n)) = F'(T(v_1), \ldots T(v_n))$$

If T maps all the characterizing operations of Q to corresponding ones in Q', we say that "T *maps* Q *into* Q' ".

T *preserves* datatype Q if T maps each characterizing operation of Q to itself.

NOTE – As an example of how the above concepts can be used, assume that we need to define a translation procedure TF that replaces all boxes in a value V with new ones while preserving the sharing structure within V TF invoked on input sequence V operates as follows:

1. Compute the set
   $\{B_1,...B_n\}$ = Assoc (V) $\bigcap$ Boxes
2. For i = 1 to n,
   C_i = create()
3. Let Z be a function Z: Value → Value satisfying
   For any box B_i, $Z(B_i) = C_i$
   Z preserves all aggregate datatypes except Box
   Z is an identity on all non-aggregate datatypes
4. Finally, set TF(V) = Z(V).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7 Interface Definition Notation

The Interface Definition Notation is the means defined in this International Standard for specifying declarations for procedures, procedure parameters, datatypes, and attributes. This concrete notation supports the datatypes defined in ISO/IEC 11404 - Language-Independent Datatypes.

NOTE – For a language processor conform to this clause of this International Standard (see 4), it is necessary that a binding be specified between the procedure calling mechanism for that language processor and the IDN defined in this clause. This binding will need to incorporate inward and/or outward mappings for the datatypes of that language to the datatypes defined in ISO/IEC 11404 - Language-Independent Datatypes, depending on the mode of conformity (client, server, or both) that is required.

7.1 Definitional Conventions

7.1.1 Character Set

```
letter      a b c d e f g h i j k l m n o p q r s t u v w x y z
digit       0 1 2 3 4 5 6 7 8 9
special     ()          {}       <>                .           ,
            (parentheses) (braces) (angle-brackets) (full stop) (comma)
            :       ;          =        /         *           -
            (colon) (semicolon) (equals) (solidus) (asterisks) (minus)
hyphen      -
apostrophe  '
quote       "
escape      !
space
```

The character space is required to be bound to the "space" member of ISO/IEC 10646-1:1993, but it only has meaning within character-literals and string-literals.

A *bound-character* is defined to be a letter, digit, hyphen, special, apostrophe, space, or quote. A bound-character is required to be associated with the member having the corresponding symbol in any character-set derived from ISO/IEC 10646-1:1993, except that no significance is attached to the "case" of the letters.

A bound-character and the escape character are required in any implementation to be associated with particular members of the implementation character set

An *added-character* is a character not defined in this International Standard. An added-character is any other member of the implementation character-set which is bound to the member having the corresponding symbol in an ISO/IEC 10646-1:1993 character set.

7.1.2 Formal Syntax

This International Standard defines a formal representation for datatype declaration and identification. The following notation, derived from Backus-Naur form, is used in defining that formal

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

representation. In this clause, the word *marks* is used to refer to the characters used to define the formal mechanism, while the word *character* is used to refer to the characters used in forming procedure and datatype declarations and identifications.

A terminal symbol is a sequence of characters delimited by two occurrences of the quotation-mark ("), the first of which precedes the first character in the terminal symbol, and the second of which follows the last character in the terminal symbol. A terminal symbol represents the occurrence of a sequence of characters.

A non-terminal symbol is a sequence of marks, each of which is either a letter or the hyphen mark (-), terminated by the first mark which is neither a letter nor a hyphen. A non-terminal symbol represents any sequence of terminal symbols which satisfies the production for that non-terminal symbol. For each non-terminal symbol there is exactly one IDN production. Non-terminal symbols are highlighted within the text of this International Standard by italics.

A repeated sequence is a sequence of terminal and/or non-terminal symbols enclosed between an open-brace mark ({) and a close-brace mark (}). The sequence of symbols so enclosed is permitted to occur any number of times at the place where the repeated sequence occurs, but is not required to occur at all.

An optional sequence is a sequence of terminal and/or non-terminal symbols enclosed between and open-bracket ([) and a close-bracket (]). The sequence of symbols so enclosed is permitted to occur once at the place where the optional sequence occurs, but is not required to occur at all.

An alternative sequence is a sequence of terminal and/or non-terminal symbols preceded by the vertical stroke mark (|) and followed by either a vertical stroke mark or a full-stop mark (.). The sequence of symbols so delimited is permitted to occur instead of the sequence of symbols preceding the first vertical stroke.

A production defines the validsequences of symbols which a non-terminal symbol represents. A simple production has the form:

```
non-terminal-symbol = valid-sequence.
```

where valid-sequence is any sequence of terminal symbols, non-terminal symbols, optional sequences, repeated sequences and alternative sequences. The equal-sign mark (=) separates the non-terminal symbol being defined from the valid-sequence which represents its definition. The full-stop mark terminates the valid-sequence.

7.1.3 Whitespace

A sequence of one or more space characters, except within a character-literal or string-literal, shall be considered whitespace. Any use of this International Standard may define any other characters or sequences of characters to be whitespace, such as horizontal and vertical tabulators, end of line and page indicators, etc.

A comment is any sequence of characters beginning with the sequence "/*" and terminating with the first occurrence thereafter of the sequence "*/". Every character of a comment shall be considered whitespace.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Any two objects which occur consecutively may be separated by whitespace, without affect on the interpretation of the syntactic construction. Whitespace shall not appear within lexical objects.

7.2 Interface Type Declarations

```
interface-type = "interface" [interface-synonym ":"]
                 [interface-identifier] "begin" interface-body "end".
interface-synonym = identifier.
interface-identifier = object-identifier.
interface-body = {import} {declaration ";"}.
declaration = value-decl | type-decl | procedure-decl | termination-decl.
```

NOTE – An interface type definition contains the declaration of various interface entities, such as constants, datatypes, components of generated types (e.g., fields of a record), etc. These declarations associate an identifier with the interface entity given in the declaration. The usage of this identifier is called its defining occurrence. When this identifier is used elsewhere in the interface type definition, it refers to the entity associated with its defining occurrence. In order to avoid ambiguity as to which entity a reference identifier refers to, rules governing the uniqueness of defining identifiers and rules governing how to resolve reference identifiers are provided in the appropriate clauses.

The *interface-synonym* in the *interface-type* declaration is an optional human readable name for an interface type. The *interface-identifier* of this production is an *object-identifier* that uniquely identifies the interface type definition.

All *interface-synonyms* shall be unique within the immediately containing *interface-type*.

7.2.1 Type references

In an interface body, an *identifier* in an interface type definition used to refer to a *type-specifier* is called a *type-reference* (see 7.7).

A *type-reference* matches a *type-decl* if the *type-identifier* of the *type-decl* is the same as the *identifier* component of the *type-reference*. The following rules govern the use of *type-references* within an *interface-type*.

If the *interface-synonym* component of the *type-reference* is absent then the *type-reference* shall either match a *type-decl* in the immediately containing *interface-type* or match a *type-decl* which is imported into the immediately containing *interface-type* (either explicitly as an *import-symbol* or implicitly by importing an entire interface type definition). If the *type-reference* matches a *type-decl* in the immediately containing *interface-type*, then it refers to the immediately contained *type-specifier* of that *type-decl*. Otherwise, the *type-reference* shall match at most one imported *type-decl*, and the *type-reference* refers to the immediately contained *type-specifier* of that *type-decl*.

NOTE – If the *type-identifier* of an imported *type-decl* is the same as a *type-identifier* defined in the immediately containing *interface-type* or is the same as a *type-identifier* of a *type-decl* imported from a different interface type definition, then it may only be referenced using its associated *interface-synonym*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

If the *interface-synonym* component of the *type-reference* is present then the *type-reference* shall match a *type-decl* in the interface type definition denoted by the *interface-synonym*. The *type-reference* refers to the immediately contained *type-specifier* of this *type-decl*.

7.2.2 Value References

In an interface body, an *identifier* in an interface type definition used to refer to a value is called a *value-reference* A *value-reference* shall refer to either:

(a) *value-expression* used in an *value-decl*; or

(b) an *enumeration-identifier*; or

(c) a *field* within a *record-type*; or

(d) a formal *parameter* of a *procedure-decl*, *procedure-type*, or *termination-decl*; or

(e) a *return-arg* within a *procedure-decl* or *procedure-type*; or

(f) a *formal-value-parm* of a *parameterized-type-decl*.

The value of a *value-reference* may be known statically, if it refers to a *value-expression* or *enumeration-identifier*. Otherwise, it is determined at the time of procedure invocation or termination.

7.3 Import Declarations

```
import = "imports" ["("import-symbol-list")"] "from"
                   [interface-synonym ":"] object-identifier.
import-symbol-list = import-symbol {"," import-symbol}.
import-symbol = identifier.
```

The *import* declaration shall be used to allow the current *interface-body* to reference *identifiers* defined in other interface type declarations. The *object-identifier* in the *import* statement is the *interface-identifier* of the interface type definition in which the symbols are defined. The *interface-synonym* in the *import* production, if present, may be used within the scope of the current *interface-body* as a prefix when referencing the imported symbol.

Each *import-symbol* shall be an *identifier* that is defined by a *value-decl*, a *type-decl*, a *procedure-decl*, or a *termination-decl* in the *interface-body* of the interface type definition denoted by the *object-identifier* in the *import* statement. Only those *import-symbols* that appear in the *import-symbol-list* shall be used within the scope of the current *interface-body*. The meaning associated with the *import-symbol* is that which it has in its defining interface type definition. If no *import-symbol-list* is present, then the entire interface is imported. This is equivalent to explicitly importing (as an *import-symbol*) every identifier defined by a *value-decl*, *type-decl*, *procedure-decl*, and *termination-decl* in the referenced interface type definition.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.4 Value Declarations

```
value-decl = "value" value-identifier ":"
                     constant-type-spec "=" value-expression.
value-identifier = identifier.

constant-type-spec = integer-type | real-type | character-type |
                     boolean-type | enumerated-type | state-type |
                     ordinal-type | time-type | bit-type | rational-type |
                     scaled-type | complex-type.

value-expression = value-reference | procedure-reference |
                   integer-literal | real-literal | character-literal |
                   boolean-literal | enumerated-literal | state-literal |
                   ordinal-literal | time-literal | bit-literal |
                   rational-literal | scaled-literal | complex-literal |
                   void-literal.
```

A *value-decl* declares an *identifier* to be equal to a constant value of a given datatype. This *identifier* may then be used wherever a *value-expression* of that datatype may be used in the interface type definition (e.g., in declaring the bounds of an array).

All *value-identifiers* shall be unique within the immediately containing *interface-type*. A value expression is either a *literal* (immediate value) of the specified type or a *value-reference*. This *value-reference* shall refer to a *value-expression* declared in another *value-decl* or to an enumeration literal (if the specified datatype is an enumeration).

7.5 Datatype Declarations

Paragraphs in this clause which refer for a formal interpretation to ISO/IEC 11404 are included for completeness and assistance to the reader, and are considered to be informative parts of this International Standard.

```
type-decl = "type" type-identifier "=" type-specifier |
            parameterized-type-decl.
type-identifier = identifier.
```

A datatype declaration declares an *identifier* to be a specific type. This *identifier* may then be used wherever a *type-specifier* may be used in the interface (e.g., to define the datatype of a parameter in a procedure declaration). The syntax and semantics of the *parameterized-type-decl* is given in clause 7.6.

All *type-identifiers* shall be unique within the immediately containing *interface-type*.

The semantics of all datatypes given in this document are consistent with the semantics of the datatypes as defined in ISO/IEC 11404 - Language-Independent Datatypes.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
type-specifier = primitive-datatype | generated-datatype | defined-datatype.
defined-datatype = type-reference [subtype-spec].
```

The *type-reference* in the *defined-datatype* production shall refer to a *type-specifier*. The *type-identifier* defined in the immediately containing *type-decl* is a synonym for the *type-specifier* referred to by the *defined-datatype*. If the *type-reference* refers to an *integer-type*, *real-type*, or an *enumerated-type* then an optional *subtype-spec* may be included. If the *type-reference* refers to a *real-type* and a *subtype-spec* is included, that *subtype-spec* shall only include a single *range* of real values.

7.5.1 Primitive Datatypes

```
primitive-datatype = integer-type | real-type | character-type |
                     boolean-type | enumerated-type | octet-type |
                     procedure-type | state-type | ordinal-type |
                     time-type | bit-type | rational-type |
                     scaled-type | complex-type | void-type.
```

7.5.1.1 Integer

Integer is the mathematical datatype comprising the exact integral values. Syntax:

```
integer-type = "integer".
integer-literal = ["-"]digit{digit}.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.2 The real datatype

Real is a family of datatypes which are computational approximations to the mathematical datatype comprising the "real numbers". Specifically, each real datatype designates a collection of mathematical real values which are known to certain applications to some finite precision and must be distinguishable to at least that precision in those applications. Syntax:

```
real-type = "real" [ "(" radix "," factor ")" ].
radix = value-expression.
factor = value-expression.
real-literal = integer-literal ["."digit{digit}] [["-"] "E" digit{digit}].
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.5.1.3 The character datatype

Character is a family of datatypes whose value spaces are character-sets. Syntax:

```
character-type = "character" ["(" repertoire-list")"].
repertoire-list = repertoire-identifier {"," repertoire-identifier}.
repertoire-identifier = value-expression.
character-literal = "'"character"'".

character =
            The value of character shall be any character drawn
            from the character set identified by the repertoire
            identifier in the production character-type, or from the
            default character set if the repertoire identifier is
            absent.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.4 The boolean datatype

Boolean is the mathematical datatype associated with two-valued logic. Syntax:

```
boolean-type = "boolean".
boolean-literal = "true" | "false".
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.5 The enumerated datatype

Enumerated is a family of datatypes, each of which comprises a finite number of distinguished values having an intrinsic ordering. Syntax:

```
enumerated-type = "enumerated" "(" enumerated-value-list ")".
enumerated-value-list = enumerated-literal {"," enumerated-literal}.
enumerated-literal = identifier.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.6 The octet datatype

```
octet-type = "octet".
```

According to ISO/IEC 11404 - Language-Independent Datatypes, the octet datatype is the derived datatype: array (1..8) of (bit).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.5.1.7 The procedure datatype

Procedure generates a datatype, called a procedure datatype, each of whose values is an operation on values of other datatypes, designated the argument datatypes. That is, a procedure datatype comprises the set of all operations on values of a particular collection of datatypes. All values of a procedure datatype are conceptually atomic. Syntax:

```
procedure-type = "procedure" "(" [argument-list] ")"
                 ["returns" "(" return-argument ")"]
                 ["raises" "(" termination-list ")" ].
argument-list = argument-declaration {"," argument-declaration}.
argument-declaration = direction argument.
direction = "in" | "out" | "inout".
argument = argument-name ":" ["restricted"] argument-type.
argument-type = type-specifier.
argument-name = identifier.
return-argument = [argument-name ":"] argument-type.
termination-list = termination-reference {"," termination-reference}.
termination-reference = [interface-synonym "::"] identifier.
```

A procedure-declaration associates one name with a procedure-type, as part of the *interface-type* association (see 6.9).

All *termination-references* shall be unique within the immediately containing *interface-type*.

7.5.1.7.1 Procedure parameters

An *argument-type* may designate any datatype. The *argument-names* of *arguments* in the *argument-list* shall be distinct from each other and from the *argument-name* of the *return-argument*, if any. The *termination-references* in the *termination-list*, if any, shall be distinct.

7.5.1.7.2 Procedure values

The values of a procedure-type are procedure closures, as defined in clause 6.6. An *argument* in the *argument-list* is said to be an input argument if its *argument-declaration* contains the direction "in" or "inout". The input space is the cross-product of the value spaces of the datatypes designated by the *argument-types* of all the input arguments. An argument is said to be a result argument if it is the *return-argument* or it appears in the *argument-list*and its *argument-declaration* contains the direction "out" or "inout". The normal result space is the cross-product of the value spaces of the datatypes designated by the *argument-types* of all the result arguments, if any, and otherwise the value space of the void datatype. When there is no *termination-list*, the result space of the procedure datatype is the normal result space, and every value p of the procedure datatype is a function of the mathematical form:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

$$p: \quad I_1 * I_2 * \ldots * I_n \rightarrow R_p * R_1 * R_2 * \ldots * R_m$$

where I_k is the value space of the argument datatype of the *k*th input argument, R_k is the value space of the argument datatype of the *k*th result argument, and R_p is the value space of the return-argument.

When a *termination-list* is present, each *termination-reference* is associated, by some *termination-declaration*, with an alternative result space which is the cross-product of the value spaces of the datatypes designated by the *argument-types* of the *arguments* in the *termination-argument-list*. Let A^j be the alternative result space of the *j*th termination. Then:

$$A^j = E^j_1 * E^j_2 * \ldots * E^j_{mj}$$

where E^j_k is the value space of the argument datatype of the *k*th argument in the *termination-argument-list* of the *j*th termination. The normal result space then becomes the alternative result space associated with normal termination (A^0), modelled as having *termination-identifier* "*normal". Consider the *termination-references*, and "*normal", to represent values of an unspecified state datatype S_T. Then the result space of the procedure datatype is:

$$S_T * (A^0 \mid A^1 \mid A^2 \mid \ldots \mid A^N),$$

where A^0 is the normal result space and A^k is the alternative result space of the *k*th termination; and every value of the procedure datatype is a function of the form:

$$p: \quad I_1 * I_2 * \ldots * I_n \rightarrow S_T * (A^0 \mid A^1 \mid A^2 \mid \ldots \mid A^N).$$

Any of the input space, the normal result space and the alternative result space corresponding to a given *termination-identifier* may be empty. An empty space can be modelled mathematically by substituting for the empty space the value space of the datatype Void.

The value space of a procedure datatype conceptually comprises all operations which conform to the above model, i.e. those which operate on a collection of values whose datatypes correspond to the input argument datatypes and yield a collection of values whose datatypes correspond to the argument datatypes of the normal result space or the appropriate alternative result space. The term "corresponding" in this regard means that to each argument datatype in the respective product space the "collection of values" shall associate exactly one value of that datatype. When the input space is empty, the value space of the procedure datatype comprises all niladic operations yielding values in the result space. When the result space is empty, the mathematical value space contains only one value, but the value space of the computational procedure datatype may contain many distinct values which differ in their effects on the "real world", i.e. physical operations outside of the information space. Value syntax:

```
procedure-declaration =
          "procedure" procedure-identifier "(" [argument-list] ")"
                        ["returns" "("return-argument")"]
                        ["raises" "("termination-list")"].
procedure-identifier = identifier.
procedure-reference = procedure-identifier.
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A *procedure-declaration* declares the *procedure-identifier* to refer to a (specific) value of the procedure datatype whose *type-specifier* is identical to the *procedure-declaration* after deletion of the *procedure-identifier*.

7.5.1.7.3 Procedure subtypes

For two procedure datatypes P and Q:

- P is said to be formally compatible with Q if their *argument-lists* are of the same length, the *direction* of each *argument* in the *argument-list* of P is the same as the corresponding *argument* in the *argument-list* of Q, both have a *return-argument* or neither does, and the *termination-lists* of P and Q, if present, contain the same *termination-references*.
- If P is formally compatible with Q, and for every result argument of Q, the argument datatype of the corresponding argument of P is a (not necessarily proper) subtype of the argument datatype of the argument of Q, then P is said to be a result-subtype of Q. If the return argument datatype and all of the argument datatypes in the *argument-list* of P and Q are identical (none are proper subtypes), then each is a result-subtype of the other.
- If P is formally compatible with Q, and for every input argument of Q, the argument datatype of the corresponding argument of P is a (not necessarily proper) subtype of the argument datatype of the argument of Q, then Q is said to be an input-subtype of P. If all of the input argument datatypes in the *argument-lists* of P and Q are identical (none are proper subtypes), then each is an input-subtype of the other.

7.5.1.8 The state datatype

State is a family of datatypes, each of which comprises a finite number of distinguished but unordered values with no characterizing operations, except Equal. Syntax:

```
state-type = "state" "(" state-value-list ")".
state-value-list = state-literal {"," state-literal}.
state-literal = identifier.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.9 The ordinal datatype

Ordinal is the datatype of the ordinal numbers, as distinct from the quantifying numbers (datatype Integer). Ordinal is the infinite enumerated datatype. Syntax:

```
ordinal-type = "ordinal".
ordinal-literal = digit {digit}.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.5.1.10 The time datatype

Time is a family of datatypes whose values are points in time to various common resolutions: year, month, day, hour, minute, second, and fractions thereof. Syntax:

```
time-type = "time" "(" time-unit ["," radix "," factor]")".

time-unit = "year" | "month" | "day" | "hour" | "minute" | "second" |
            parametric-value.

time-literal = digit{digit} ["."digit{digit}].
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.11 The bit datatype

Bit is the datatype representing the finite field of two symbols designated "0", the additive identity, and "1", the multiplicative identity. Syntax:

```
bit-type = "bit".
bit-literal = "0" | "1".
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.12 The rational datatype

Rational is the mathematical datatype comprising the "rational numbers". Syntax:

```
rational-type = "rational".
rational-literal = ["-"] digit{digit} ["/" digit{digit}].
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.13 The scaled datatype

Scaled is a family of datatypes whose value spaces are subsets of the rational value space, each individual datatype having a fixed denominator, but the scaled datatypes possess the concept of approximate value. Syntax:

```
scaled-type = "scaled" "(" radix "," factor ")".
scaled-literal = ["-"] digit{digit} [fraction].
fraction = "." digit{digit}.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.5.1.14 The complex datatype

Complex is a family of datatypes, each of which is a computational approximation to the mathematical datatype comprising the "complex numbers". Specifically, each complex datatype designates a collection of mathematical complex values which are known to certain applications to some finite precision and must be distinguishable to at least that precision in those applications. Syntax:

```
complex-type = "complex" ["(" radix "," factor ")"].
complex-literal = "(" real-part "," imaginary-part ")".
real-part = real-literal.
imaginary-part = real-literal.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.1.15 The void datatype

Void is the datatype representing an object whose presence is syntactically or semantically required, but carries no information in a given instance. Syntax:

```
void-type = "void".
void-literal = "nil".
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.2 Generated datatypes

```
generated-datatype = record-type
                   | choice-type
                   | array-type
                   | pointer-type.
```

7.5.2.1 The record datatype

Record generates a datatype, called a record datatype, whose values are heterogeneous aggregations of values of component datatypes, each aggregation having one value for each component datatype, keyed by a fixed "field-identifier". Syntax:

```
record-type = "record" "of" "("field-list")".
field-list = field {"," field}.
field = field-identifier ":" field-type.
field-identifier = identifier.
field-type = type-specifier.
```

All *field-names* shall be unique within their immediately containing *record-type* or *choice-type*.

The interpretation of the syntax is formally defined in ISO/IEC 11404.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.5.2.2 The choice datatype

Choice generates a datatype called a choice datatype, each of whose values is a single value from any of a set of alternative datatypes. The alternative datatypes of a choice datatype are logically distinguished by their correspondence to values of another datatype, called the tag datatype.

```
choice-type = "choice" "("tag-type")" "of" "("alternative-list")".
tag-type = type-specifier.
alternative-list = alternative { "," alternative} [default-alternative].
alternative = tag-value-list ":" alternative-type.
default-alternative = "default" ":" alternative-type.
alternative-type = type-specifier.
tag-value-list = select-list.
select-list = select-item {"," select-item}.
select-item = value-expression | select-range.
select-range = lowerbound ".." upperbound.
lowerbound = value-expression | "*".
upperbound = value-expression | "*".
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

7.5.2.3 The array datatype

Array generates a datatype, called an array datatype, whose values are associated between the product space of one or more finite datatypes, designated the index datatypes, and the value space of the element datatype, such that every value in the product space of the index datatypes associates to exactly one value of the element datatype. Syntax:

```
array-type = "array" "("index-type-list")" "of" "("element-type")".
index-type-list = index-type {"," index-type}.
index-type = type-specifier | index-lowerbound ".." index-upperbound.
index-lowerbound = value-expression.
index-upperbound = value-expression.
element-type = type-specifier.
```

The interpretation of the syntax is formally defined in ISO/IEC 11404.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

7.5.2.4 The pointer datatype

Pointer generates a datatype, called a pointer datatype, each of whose values constitutes a means of reference to values of another datatype, designated the element datatype. The values of a pointer datatype are atomic. Syntax:

```
pointer-type = "pointer" "to" "("element-type")".
```

Pointer is a type-generator which generates a primitive datatype each of whose values constitutes a means of reference to values of another datatype, designated the *element-type*. The values of a pointer datatype are boxes, as defined in clause 6.7.

A pointer with the "restricted" attribute is a pointer that never has the null label and is neither statically nor dynamically aliased with any other pointer. Restricted pointers can be supported efficiently; however, due to the optimized protocol it is impossible to determine whether the label of an inout restricted pointer was changed as a result of executing the server procedure.

A pointer value is said to be statically aliased at a procedure invocation if there is more that one Box which contains it among the generalized associates of the invocation association at initiation.

> NOTE – Static aliasing is a property of the closure, while dynamic aliasing is a property of the invocation. The above definitions make the assumption that a formal parameter becomes a Box in the invocation association containing the actual parameter value. Since this is not actually required, the notion Box must be extended to include the instantiation of the formal/actual parameter bindings for the purposes of the above definition only.

7.5.3 Subtypes

```
subtype-spec = "select" "("select-element {"," select-element}")".
select-element = value-expression | range.
range = lower-bound ".." upper-bound | ".." upper-bound | lower-bound "..".
lower-bound = value-expression.
upper-bound = value-expression.
```

A *subtype-spec* consists of a list of elements, where each element is either a *value-expression* of the specified datatype or a *range* of values of the specified type. The *value-expressions* that occur in a *subtype-spec* must refer to either a literal (immediate value), an enumeration literal, or to a *formal-value-parm*.

7.6 Parameterized Types

```
parameterized-type-decl =
     "type" type-identifier "("formal-value-parms")" "=" type-specifier.
formal-value-parms = formal-value-parm {"," formal-value-parm}.
formal-value-parm = identifier ":" value-param-type-spec.
value-parm-type-spec = type-specifier.
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A *parameterized-type-decl* introduces a partial specification of a datatype. It associates a *type-identifier* and a set of formal parameters, called *formal-value-parms*, with a *type-specifier*. Each *formal-value-parm* is itself an *identifier* that can be referenced from within the *type-specifier*. References to these *formal-value-parms* can only be used in place of *value-expressions* within the *type-specifier* (e.g., in place of an array bound).

Each *formal-value-param* has a *value-param-type-spec* associated with it, specifying the datatype of the *formal-value-param*. This type shall be a datatype that a *value-expression* may have in an interface type definition.

The *type-identifier* introduced by a *parameterized-type-decl* can be used anywhere a *type-specifier* can be used in the interface, as long as actual values are provided for the *formal-value-parms* of the *parameterized-type-spec*. Hence, whenever this *type-identifier* is referenced, it shall be referenced as a *parameterized-type-reference*.

A *parameterized-type-decl* shall not directly reference itself (via a *parameterized-type-reference*) nor shall it reference itself indirectly (via a *parameterized-type-reference* to a different parameterized type the directly or indirectly references this parameterized type).

All *formal-value-parms* shall be unique within the immediately containing *parameterized-type-decl*.

7.7 Identifiers

```
object-identifier = "{"ObjectIdComponent {ObjectIdComponent}"}".
ObjectIdComponent = identifier | digit | identifier "("digit {digit}")".
```

The syntax for *object-identifier* is that of an ASN.1 ObjectIdentifierValue, as defined in ISO 8824.

```
type-reference = [interface-synonym "::" ] identifier |
                 parameterized-type-reference.

parameterized-type-reference = [interface-synonym "::"]
                                    identifier "("actual-value-parm
                                    {"," actual-value-parm}")".

actual-value-parm = value-reference.
```

Wherever a *parameterized-type-reference* is used in the *interface-type*, it shall reference the *type-specifier* of a *parameterized-type-decl*. An *actual-value-parm* must be supplied for each *formal-value-parm* of the *parameterized-type-decl*. The datatype of an *actual-value-parm* must be the same as the datatype of the corresponding *formal-value-parm*. The semantics of the resulting *type-specifier* is that obtained by replacing each *formal-value-parm* reference within the *type-specifier* by the corresponding *actual-value-parm*.

```
value-reference = [interface-synonym "::"] identifier {"." identifier}.
identifier = letter {pseudo-letter}.
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|
         "P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|
         "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|
         "p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".

pseudo-letter = letter | digit | underline.
digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
underline = "_".
```

7.7.1 Value references to fields

A *value-reference* matches a *field* if:

(a) the *field* is immediately contained within a *record-type* R; and

(b) the *value-reference* is contained within R; and

(c) the first identifier component of the *value-reference* is the same as the *field-name* of the *field*; and

(d) the *value-reference* is not contained within a *procedure-type* that is contained within R; and

(e) there is no *record-type* R2 such that R2 is contained within R, and a), b), c) and d) are true when substituting R2 for R.

If a *value-reference* matches a *field*, then the first *identifier* of the *value-reference* refers to that *field*. If the ith *identifier* of a *value-reference* refers to a *field* and the *value-reference* consists of more than i *identifiers*, then the *field* that the ith *identifier* refers to shall be a *record-type*, and the (i+1)th *identifier* of the *value-reference* shall be the same as a *field-name* of this *record-type*. The (i+1)th *identifier* of the *value-reference* refers to the *field* associated with the *field-name*. If the ith *identifier* of a *value-reference* refers to a *field* and the *value-reference* consists of exactly i *identifiers*, then the *value-reference* refers to this *field*.

7.7.2 Value references to parameters, return-args, or to fields contained within them

A *value-reference* matches a *parameter* (*return-arg*) if:

(a) the *value-reference* does not match a *field*; and

(b) the *parameter* (*return-arg*) is immediately contained within a *procedure-decl* or *procedure-type* P; and

(c) the *value-reference* is contained within P; and

(d) the first *identifier* component of the *value-reference* is the same as the *parameter-name* (*identifier*) of the *parameter* (*return-arg*); and

(e) the *value-reference* is not contained within a *procedure-type* (distinct from P) that is contained within P.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

If a *value-reference* matches a *parameter* (*return-arg*) and the *value-reference* consists of a single *identifier*, then the *value-reference* refers to that *parameter* (*return-arg*). Otherwise, the *parameter* (*return-arg*) must be a *record-type* and *value-reference* shall refer to a *field*, following the rules given in clause 7.7.1.

7.7.3 Value references to formal-value-parms

A *value-reference* matches a *formal-value-parm* if:

(a) the *value-reference* does not match a *field*, a *parameter*, nor a *return-arg*; and

(b) the *formal-value-parm* is immediately contained within a *parameterized-type-decl*; and

(c) the *value-reference* is contained within the *type-specifier* of this *parameterized-type-decl* and is the same as the *formal-value-parm*.

If a *value-reference* matches a *formal-value-parm* then it refers to that *formal-value-parm*.

7.7.4 Value references to value-expressions

A *value-reference* matches a *value-decl* if the *value-identifier* of the *value-decl* is the same as the *identifier* component of the *value-reference*.

If the *interface-synonym* component of the *value-reference* is absent, and the *value-reference* matches a *value-decl* in the immediately containing *interface-type*, and the *value-reference* does not match a *field*, a *parameter*, a *return-arg*, nor a *formal-value-arg*, then the *value-reference* refers to the immediately contained *value-expression* of that *value-decl*. Otherwise, if the *interface-synonym* component of the *value-reference* is absent, and the *value-reference* matches exactly one imported *value-decl*, and the *value-reference* does not match a *field*, a *parameter*, a *return-arg*, nor a *formal-value-parm*, then the *value-reference* refers to the immediately contained *value-expression* of that *value-decl*.

NOTE – If the *value-identifier* of an imported *value-decl* is the same as a *value-identifier* defined in the immediately containing *interface-type* or is the same as a *value-identifier* of a *value-decl* imported from a different interface type definition, then it may only be referenced using its associated *interface-synonym*.

If the *interface-synonym* component of the *value-reference* is present and *value-reference* matches a *value-decl* in the interface type definition denoted by the *interface-synonym*, then the *value-reference* refers to the immediately contained *value-expression* of this *value-decl*.

7.7.5 Value references to enumeration-identifiers

When the *type-identifier* component of the *value-reference* is present, a *value-reference* matches an *enumeration-identifier* of an *enumerated-type* if the *type-identifier* of the *value-reference* is the same as an *enumeration-identifier* of the *enumerated-type*. If the *type-identifier* is not present, a *value-reference* matches an *enumeration-identifier* of an *enumerated-type* if the *identifier* component of the *value-reference* is the same as an *enumeration-identifier* of the *enumerated-type*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

If the *interface-synonym* component of the *value-reference* is absent, and the *value-reference* matches exactly one *enumeration-identifier* in the immediately containing *interface-type*, and the *value-reference* does not match a *field*, a *parameter*, a *return-arg*, a *formal-value-parm*, nor a *value-expression*, then the *value-reference* refers to the matching *enumeration-identifier*. Otherwise, if the *interface-synonym* component of the *value-reference* is absent, and the *value-reference* matches exactly one imported *enumeration-identifier*, and the *value-reference* does not match a *field*, a *parameter*, a *return-arg*, a *formal-value-parm*, nor a *value-expression*, then the *value-reference* refers to the imported matching *enumeration-identifier*.

If the *interface-synonym* component of the *value-reference* is present, and the *value-reference* matches exactly one *enumeration-identifier* in the interface type definition denoted by the *interface-synonym*, and the *value-reference* does not match a *value-expression* in the definition denoted by the *interface-synonym*, then the *value-reference* refers to the matching *enumeration-identifier*.

7.7.6 Termination references

The rules governing the resolution of *termination-references* are identical to the rules governing the resolution of *type-references*.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex A
(informative)

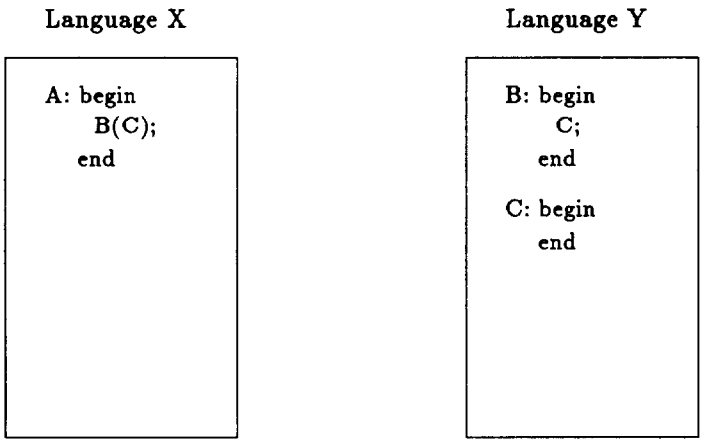
Procedure Parameters

The syntax for the language-independent calling mechanism allows for a procedure to be a parameter of another procedure. There are three different cases that result from the procedure parameters feature.

A.1 LIPC Reference / Local Access

In this case, procedure A in language X calls procedure B in language Y and passes to procedure B a pointer to procedure C which is also in language Y. There shall exist a way for language X to reference procedure C in order to generate a pointer to pass to procedure B. This reference to C shall be referred to as the lipc-reference. After B has begun execution, it will eventually call C, but this is simply a local call therefore no lipc-access is necessary.

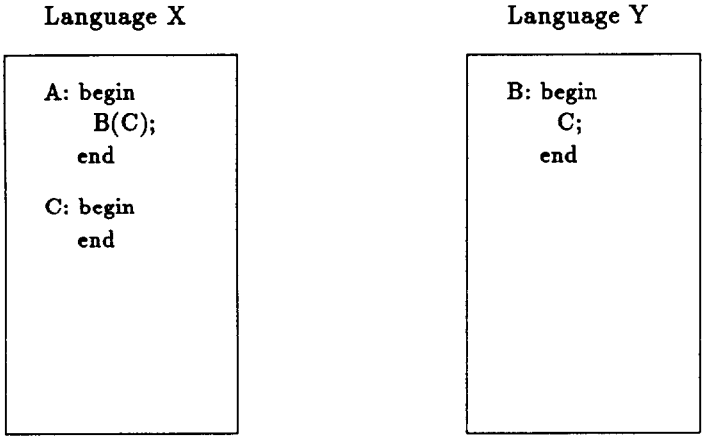
NOTE – Procedure B must understand how to call procedure C "locally" based on the lipc-reference information it was passed.



A.2 LIPC Reference / LIPC Access

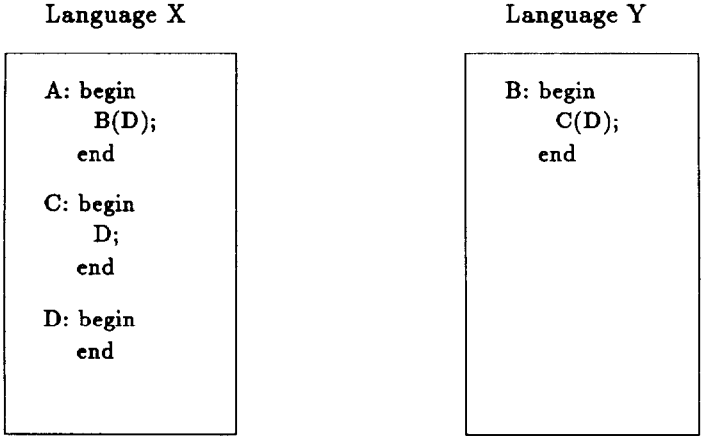
In this case, procedure A in language X calls procedure B in language Y and passes to procedure B a pointer to procedure C which which is in language X. Eventually, B will call C and in this case the call to C must use lipc-access since the call crosses the boundary. In addition to this for B to call C, it must have the lipc-reference of C. This information is obtained from that which was passed from procedure A.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale



A.3 Local Reference / Local Access

In this case, procedure A in language X calls procedure B in language Y and passes to procedure B a pointer to routine D in language X. Eventually, B will call procedure C in language X and pass to procedure C the pointer to routine D. C will then call D, but in this case both the reference and access of D by C are local. Therefore it is not necessary for the pointer information describing D to be a lipc-reference, but it must be in a form that allows the transformation to B's environment and back to its original state.



Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

Annex B
(informative)

Interface Definition Notation Syntax

This annex contains the complete IDN syntax, for reference only.

Productions of the IDN — Normative text page

actual-value-parm = value-reference. 41

alternative = tag-value-list ":" alternative-type. 39

alternative-list = alternative { "," alternative} [default-alternative]. 39

alternative-type = type-specifier. 39

argument = argument-name ":" ["restricted"] argument-type. 34

argument-declaration = direction argument. 34

argument-list = argument-declaration {"," argument-declaration}. 34

argument-name = identifier. 34

argument-type = type-specifier. 34

array-type = "array" "("index-type-list")" "of" "("element-type")". 39

bit-literal = "0" | "1". 37

bit-type = "bit". 37

boolean-literal = "true" | "false". 33

boolean-type = "boolean". 33

character = 33

The value of character shall be any character drawn from the character set identified by the repertoire identifier in the production character-type, or from the default character set if the repertoire identifier is absent.

character-literal = "'"character"'". 33

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
character-type = "character" ["(" repertoire-list")"].                    33

choice-type = "choice" "("tag-type")" "of" "("alternative-list")".        39

complex-literal = "(" real-part "," imaginary-part ")".                   38

complex-type = "complex" ["(" radix "," factor ")"].                      38

constant-type-spec = integer-type | real-type | character-type |          31
                     boolean-type | enumerated-type | state-type |
                     ordinal-type | time-type | bit-type |
                     rational-type | scaled-type | complex-type.

declaration = value-decl | type-decl | procedure-decl | termination-decl.  29

default-alternative = "default" ":" alternative-type.                     39

defined-datatype = type-reference [subtype-spec].                         32

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".                          42

direction = "in" | "out" | "inout".                                       34

element-type = type-specifier.                                            39

enumerated-literal = identifier.                                          33

enumerated-type = "enumerated" "(" enumerated-value-list ")".             33

enumerated-value-list = enumerated-literal {"," enumerated-literal}.      33

factor = value-expression.                                                32

field = field-identifier ":" field-type.                                  38

field-identifier = identifier.                                            38

field-list = field {"," field}.                                           38

field-type = type-specifier.                                              38

formal-value-parm = identifier ":" value-param-type-spec.                 40

formal-value-parms = formal-value-parm {"," formal-value-parm}.           40

fraction = "." digit{digit}.                                              37

generated-datatype = record-type | choice-type | array-type | pointer-type.  38
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

```
identifier = letter {pseudo-letter}.                                    41

imaginary-part = real-literal.                                          38

import = "imports" ["("import-symbol-list")"] "from"                    30
            [interface-synonym ":"] object-identifier.

import-symbol = identifier.                                             30

index-lowerbound = value-expression.                                    39

index-type = type-specifier | index-lowerbound ".." index-upperbound.   39

index-type-list = index-type {"," index-type}.                          39

index-upperbound = value-expression.                                    39

integer-literal = ["-"]digit{digit}.                                    32

integer-type = "integer".                                               32

interface-body = {import} {declaration ";"}.                            29

interface-identifier = object-identifier.                               29

import-symbol-list = import-symbol {"," import-symbol}.                 30

interface-synonym = identifier.                                         29

interface-type = "interface" [interface-synonym ":"]                    29
                 [interface-identifier] "begin" interface-body "end".

letter = "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K"|"L"|"M"|"N"|"O"|  42
         "P"|"Q"|"R"|"S"|"T"|"U"|"V"|"W"|"X"|"Y"|"Z"|
         "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k"|"l"|"m"|"n"|"o"|
         "p"|"q"|"r"|"s"|"t"|"u"|"v"|"w"|"x"|"y"|"z".

lowerbound = value-expression | "*".                                   39

lower-bound = value-expression.                                         40

ObjectIdComponent = identifier | digit | identifier "("digit {digit}")".  41

object-identifier = "{"ObjectIdComponent {ObjectIdComponent}"}".        41

octet-type = "octet".                                                   33
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
ordinal-literal = digit {digit}.                                          36

ordinal-type = "ordinal".                                                 36

parameterized-type-decl =                                                 40
     "type" type-identifier "("formal-value-parms")" "=" type-specifier.

parameterized-type-reference = [interface-synonym "::"]                   41
              identifier "("actual-value-parm {"," actual-value-parm}")".

pointer-type = "pointer" "to" "("element-type")".                         40

primitive-datatype = integer-type | real-type | character-type |          32
                     boolean-type | enumerated-type | octet-type |
                     procedure-type | state-type | ordinal-type | time-type |
                     bit-type | rational-type | scaled-type | complex-type |
                     void-type.

procedure-declaration =                                                   35
          "procedure" procedure-identifier "(" [argument-list] ")"
                        ["returns" "("return-argument")"]
                        ["raises" "("termination-list")"].

procedure-identifier = identifier.                                        35

procedure-reference = procedure-identifier.                               35

procedure-type = "procedure" "(" [argument-list] ")"                      34
                 ["returns" "(" return-argument ")"]
                 ["raises" "(" termination-list ")" ].

pseudo-letter = letter | digit | underline.                               42

radix = value-expression.                                                 32

range = lower-bound ".." upper-bound | ".." upper-bound | lower-bound "..".  40

rational-literal = ["-"] digit{digit} ["/" digit{digit}].                 37

rational-type = "rational".                                               37

real-literal = integer-literal ["."digit{digit}] [["-"] "E" digit{digit}].  32

real-part = real-literal.                                                 38

real-type = "real" [ "(" radix "," factor ")" ].                          32

record-type = "record" "of" "("field-list")".                             38
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
repertoire-identifier = value-expression.                                    33

repertoire-list = repertoire-identifier {"," repertoire-identifier}.         33

return-argument = [argument-name ":"] argument-type.                         34

scaled-literal = ["-"] digit{digit} [fraction].                              37

scaled-type = "scaled" "(" radix "," factor ")".                             37

select-element = value-expression | range.                                   40

select-item = value-expression | select-range.                               39

select-list = select-item {"," select-item}.                                 39

select-range = lowerbound ".." upperbound.                                   39

state-literal = identifier.                                                  36

state-type = "state" "(" state-value-list ")".                               36

state-value-list = state-literal {"," state-literal}.                        36

subtype-spec = "select" "("select-element {"," select-element}")".           40

tag-type = type-specifier.                                                   39

tag-value-list = select-list.                                                39

termination-list = termination-reference {"," termination-reference}.        34

termination-reference = [interface-synonym "::"] identifier.                 34

time-literal = digit{digit} ["."digit{digit}].                               37

time-type = "time" "(" time-unit ["," radix "," factor]")".                  37

time-unit = "year" | "month" | "day" | "hour" | "minute" | "second" |        37
            parametric-value.

type-decl = "type" type-identifier "=" type-specifier |                      31
            parameterized-type-decl.

type-identifier = identifier.                                                31

type-reference = [interface-synonym "::" ] identifier |                      41
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

```
                parameterized-type-reference.

type-specifier = primitive-datatype | generated-datatype | defined-datatype.     32

underline = "_".                                                                  42

upperbound = value-expression | "*".                                              39

upper-bound = value-expression.                                                   40

value-decl = "value" value-identifier ":"                                         31
                     constant-type-spec "=" value-expression.

value-expression = value-reference | procedure-reference |                        31
                   integer-literal | real-literal | character-literal |
                   boolean-literal | enumerated-literal | state-literal |
                   ordinal-literal | time-literal | bit-literal |
                   rational-literal | scaled-literal | complex-literal |
                   void-literal.

value-identifier = identifier.                                                    31

value-parm-type-spec = type-specifier.                                            40

value-reference = [interface-synonym "::"] identifier {"." identifier}.          41

void-literal = "nil".                                                             38

void-type = "void".                                                               38
```


Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex C
(informative)

How to do an LIPC binding for a language

The LIPC model is based upon the familiar "client-server" concept: a client program calling a server procedure. There is a "virtual contract" between the two partners, in which the procedure (server) side agrees to provide the service (of executing the procedure), and the program (client) side agrees to provide the necessary calling information (i.e. the actual parameters) in accordance with the parameter passing methods required.

The binding of a programming language to LIPC must therefore consists of two parts, one specifying the binding when a program in the language is acting in client mode, calling an LIPC procedure, the other specifying the binding when a procedure written in the language is the subject of an LIPC call. These bindings will be expressed as requirements, respectively, on the client's LIPC service and the client side of the virtual contract, and on the server's LIPC service and the server side of the virtual contract. They must be separate and self-contained, since a particular language processor may have available only one or other LIPC service or both; i.e. it may be able to act as a client but not as a server, as a server but not as a client, or as either. However, the bindings must be consistent; an LIPC call to a procedure written in the same language must be indistinguishable from a direct call, subject only to processor-dependent variations permitted by the language standard, and any implementation constraints imposed by the particular LIPC service.

One thing to bear in mind when specifying the bindings is that languages may be conceived, designed and used with a much more integrated view of a procedure and its call than is feasible (or perhaps even desirable) in an LIPC environment. This can be reflected in the language standard, which will need to be examined in case it contains any inbuilt assumptions, that are not stated explicitly, about what it means to call a procedure. The decoupling of the client and server sides of a procedure call may need to be more complete than in the language standard, which may take a more close-coupled view. Hence aspects may be uncovered which will need to be made more explicit in the binding standard than has hitherto been customary in the language community.

The decoupling therefore needs to be accompanied by a conscious search for such implicit assumptions. Care will need to be taken, when making these explicit in the binding standard, that this process remains faithful to the view of procedure calling familiar to language users.

C.1 Linking the client and the server

Languages vary greatly in the way that the language processor is expected to recognise and locate any procedures called by a program. Strictly block-structured languages may require "declaration before use", or at least that the procedure be declared in the same block, or some surrounding block, from where the call originates. Languages with a more disjoint structure, designed for separate compilation of procedures, may assume the existence of a "link editor" - left undefined or implementation-dependent in the language standard - to make the connections. Some require explicit invocation of required libraries or modules, either within the program text, or through use of processor directives or options outside the program itself. Some make the procedure heading

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

separable from the procedure body, so the specification of the formal parameters etc. can appear explicitly in the text of the calling (client) program.

In the block-structured case, external procedures can be provided by assuming the existence of a "super-block", surrounding the outermost block of the program itself, in which all needed procedures are "declared". In the disjoint case, it is up to the (implementation-dependent) link editor to find the missing blocks, e.g. by pre-processor commands or compiler directives.

Such matters are often regarded by the responsible committee as outside the scope of the language standard. This is one situation where the language standard needs to be examined in case it contains any inbuilt assumptions, but subject to that, they can be left out of the language binding standard as well - though it should be stated explicitly that they are left out, and why. Only when the language standard explicitly addresses access to modules or procedure libraries may it be necessary to say something in the binding standard about accessing LIPC procedures.

Throughout, the essential principle to maintain is that the call of an LIPC procedure from within a client program must be indistinguishable in the program text from the call of a native language (external) procedure. The user of the client program will need to know how to access the required procedure, but that is the case for any external procedure. Nothing extra should be needed for an LIPC procedure.

If the language standard address exception handling, then regardless of how the language deals with external procedures, the binding standard should cover any exception conditions particular to LIPC calls (e.g. unable to locate external procedure, no binding available for ..... datatype parameter). Inclusion of such exception reporting may be worth considering even when the language standard itself does not address exceptions.

C.2 Client mode binding

For a language processor acting in client mode, the client LIPC service first needs to marshal the actual parameters (including the returned result, considered as an extra "out" parameter) into LIPC form. Marshalling includes, for each actual parameter, both identifying the parameter passing method, and mapping the local datatype into the corresponding LID datatype. The binding standard does not need to provide the datatype mappings, which can be established by reference to the language's LID binding. However, it does need to cite explicitly any constraints that the language imposes on the allowed datatypes for parameters (including the returned result).

> NOTE – It may be felt desirable to include "allowed extensions" of the datatypes permitted for parameters, to widen the range of LIPC procedures that can be called, e.g. to allow aggregates to be passed as parameters. However, in that case it would be logical to permit the extensions for server mode too, in which case it would seem better to include the parameter extensions in an separate addendum - optional or mandatory - to the language standard, rather than confine it to the LIPC context.

As well as datatypes of actual parameters, the binding standard should cite the parameter passing mechanisms supported by the language standard and relate them to the parameter passing modes of the LIPC standard, together with the rules for marshalling parameters of each kind. In every case it must be made explicit whether the mechanism is allowed, recommended, or required. For the purpose of LIPC binding it may be thought desirable to be stricter than in the purely local

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

case, to promote efficiency: for example, a language standard may in the general case allow either "call by reference" or "copy in copy out" for an "out" parameter, but for an LIPC call the binding could recommend, or even require, use of "copy in copy out", assuming that the server procedure itself can accept that mechanism in the virtual contract.

The client mode binding will of course also need to allow for the reverse unmarshalling of returned results (out parameters) as a result of the call.

C.3 Server mode binding

Superficially it might appear that the server mode binding would be symmetrical to the client mode binding, a kind of mirror image. However, this is not quite the case; the apparent symmetry is deceptive. It is the server side which primarily determines the virtual contract; the definition of the server procedure specifies the number and datatypes of the parameters, including the returned value if any. The server side also determines the allowable methods of parameter passing: if the client cannot pass a parameter by the required method, then the call cannot take place.

NOTE 1 – LIPC does not and cannot ensure that any valid procedure in any language can be called by a valid procedure invocation in any other language.

For a language processor acting in server mode, the server LIPC service needs to unmarshal from the incoming LIPC form into the form which would be required for the call if it were from the server's own language. This will require a datatype mapping, provided by the LID standard, and possibly also a datatype conversion.

The standard for the server language may already allow some automatic datatype conversions. For example, if a formal parameter called by value (on initiation) is of datatype **real**; then the standard may permit also an actual parameter of datatype **integer**, the integer value being converted into (say) floating point form. Therefore, if the server's LIPC server receives such an actual parameter, it can map it from LID Integer to server **integer** and pass in the value. The binding standard need say nothing about the consequent datatype conversion, since that will be handled inside the server processor just as it is were it part of an ordinary, non-LIPC call.

However, suppose that the incoming actual parameter is of LID datatype Scaled - e.g. because the client language does not support floating point or other approximate real datatype, or provides fixed-point as well as floating point? The binding standard does now need to specify whether the conversion of LID Scaled to server **real** is permitted, and if so how it is to take place.

NOTE 2 – Hence datatype conversions in an LIPC call are in general of two kinds - within the LIPC service during unmarshalling, which are specified in the binding standard, and within the server processor during procedure initialising, which are not. Some actual parameters may indeed go through conversions of both kinds. The calling client sees no difference.

The binding standard for server mode therefore needs to specify not just the LID mappings for parameter datatypes, but also, explicitly, allowed conversions from LID datatypes without direct equivalents in the server language.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

NOTE 3 – The LID binding standard may specify, or allow, or suggest such conversions, but the LIPC binding must revisit the subject in the light of the particular context of parameter passing, which may (and in many languages does) impose additional restrictions on allowed datatypes.

Mention was made, in respect of client mode binding, of the allowed parameter passing modes within the language. This holds also on the server side, of course. The binding standard should ensure that there is no inconsistency between the two.

The server mode binding will of course also need to allow for the reverse marshalling of returned results (out parameters) as a result of the call.

There are two further matters to be considered for server mode: procedure parameters, and "global variables".

C.4 Procedure parameters

If the server procedure has a formal parameter of a procedure datatype, this means that the server procedure, during execution, will call the actual procedure supplied as the actual parameter. The server procedure may, depending on the language, specify (directly or indirectly) the number and datatypes of parameters for the supplied actual procedure, or be able to treat the (formal) procedure parameter generically. Since LID procedure datatypes carry information about the number and datatypes of the procedure's parameters, the two LIPC services, on the client and server sides, will be able to communicate the necessary information through the LID mappings of the procedure parameter, to decide whether the actual procedure parameter supplied can be called by the server. If the server specifies the number and datatypes of parameters for the actual procedure, the unmarshalling can check that the incoming procedure parameter is acceptable.

In general, the call itself will be one of three kinds:

1. It may be of a procedure local to the server side
2. It may be of a procedure local to the client side
3. It may be a procedure local to neither the client nor the server, but residing on another server.

The binding standard will need to cover all three cases.

In case 1, the server can simply call the procedure normally, and continue.

In case 2, where the procedure referenced by the procedure parameter is local to the client side, the call amounts to a "reverse call" from the server side (acting in client mode for this procedure datatype parameter) to the client side (acting in server mode for the same procedure parameter). Thus procedure parameters cannot be supported unless the processors in both sides can act in both modes, and the conformity rules for the LIPC binding standard will need to reflect this.

NOTE – Note that, since both sides conform in both modes, the two LIPC services can agree parameter passing modes for calls of actual parameters of procedure datatype, as well as for the original call.

Note that the passing mode "call by value sent on request", and references in the server procedure body to pointer parameters, can all be treated as if they were reverse calls of "mini

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

procedures", but this does not need to be addressed by the LIPC binding standard. This is because supporting those features does not need full mutual procedure calling capacity, and can be left implementation-dependent - though possibly subject to constraints or conditions.

In case 3, where the procedure referenced by the procedure parameter reside on a third system, the call entails the server side to act in client mode in respect of the third system. Hence, in this case too the server side (but not the client side of the original call) must be able to act in both modes, and the conformity rules for the LIPC binding standard will need to reflect this.

C.5 Global variables

Some languages permit, in the procedure body, reference to "global variables", i.e. entities declared and specified not within the procedure body, but in some surrounding environment such as an enclosing block. In the LIPC environment, if these global variables are always provided on the server side, then they cause no problem. Problems arise only if the missing entities are assumed to be provided on the client side.

The LIPC binding standard needs to address this question. A simple solution is to say that a server procedure referencing global variables on the client side does not conform to the LIPC binding standard. However, this would preclude close-coupled cases where both the client and the server languages can reference common storage areas and the LIPC environment can support them. Another solution is to deem all such global variables to be notional additional parameters, to be passed in addition to ordinary parameters in the LIPC. These notional parameters would be passed (in virtual contract terms) in the same way as others, but the service contract between the two LIPC services would handle them via common storage, subject only to the condition that the net effect is identical to what would occur were the global variables to be replaced, in the server procedure specification, by formal parameters.

Circumstances will vary greatly, both between languages and between LIPC services and environments, and no general guidance can be given on how to address this question in the binding standard. However, if the language does allow undeclared global variables within procedure bodies, the LIPC binding standard must address it.

NOTE – The detailed LIPC model in clause 6 of what it means to call a procedure may help in deciding how to handle the question of global variables in the binding standard.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

Annex D
(informative)

LIPC IDN - RPC IDL Alignment overview

This annex compares the concepts implicit in the LIPC Interface Definition Notation (LIPC IDN) as defined in this International Standard, with the concepts implicit in the RPC Interface Definition Language (RPC IDL), as defined in Clause 4 of the RPC standard (ISO/IEC 11578).

No comparison is made of the detailed syntactic forms of the two notations.

In general, LIPC IDN provides a richer set of semantic distinctions than RPC IDL. Every RPC IDL type has an abstractly equivalent form in LIPC IDN. That is, the LIPC type describes the same set of values as the RPC type. However, RPC IDL can describe representation issues that are beyond the scope of LIPC, and RPC IDL can provide additional non-type information relevant to a particular RPC service specification.

The top level declaration in both LIPC IDN and RPC IDL is the interface declaration. All other declarations occur as part of interface declarations. Where the omission of the interface identifier in the specification does not cause any ambiguities, the interface declaration may be omitted, and the 'normal' declaration may be used as top level declaration.

D.1 Interface Declarations

The "interface" concept in LIPC and RPC are similar.

D.1.1 Attributes

LIPC IDN allows an OSI object-identifier to be supplied which uniquely identifies the interface. RPC IDL has a `uuid(X)` attribute that performs the same function, although the `X` is not an OSI object-identifier.

RPC IDL has three attributes (`version`, `endpoint`, and `local`) which pertain to the use of RPC IDL in providing an RPC service. These have no LIPC analog.

RPC IDL has a `pointer_default` attribute which allows certain pointer attributes to be omitted in the body of the interface. This is purely a notational convenience.

D.1.2 Imports Clause

Both LIPC IDN and RPC IDL permit names to be "imported" into the current interface declaration from another (preexisting) interface. This allows the names to be used in the body of the current interface.

LIPC IDN can limit the names imported from a preexisting interface (by explicitly listing the desired names). RPC IDL cannot.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

LIPC IDN permits imported names to be "qualified" with the name of the source interface. RPC IDL does not.

D.2 Other Declarations

Both LIPC IDN and RPC IDL allow interfaces to declare named types, named values, and named procedures (operations). In addition, LIPC IDN requires that termination conditions be named.

D.2.1 Type Declarations

Both LIPC IDN and RPC IDL can associate a name with any type definable in the respective notations.

LIPC IDN makes a distinction between assigning a synonym to an existing type, and defining a name for a new type. RPC IDL does not make this distinction in general, but does allow the distinction for structure types and union types. (See the "tagged_declarator" concept in RPC IDL.)

LIPC IDN allows type generators (parameterized types) to be defined (both as synonyms and new types). RPC IDL does not.

See below for a discussion of the datatypes definable in LIPC IDN and RPC IDL.

D.2.2 Value Declarations

Both LIPC IDN and RPC IDL can associate a name with a value.

LIPC IDN allows names to be given to any type of value, and provides literal notations for all types of values.

RPC IDL allows names to be given to values of a limited set of types. The permitted types are integer, boolean, character, pointer-to-character (string), and pointer-to-void.

RPC IDL permits computation of the values involved. LIPC IDN does not.

D.2.3 Procedure Declarations

Both LIPC IDN and RPC IDL can declare named procedures (operations).

In both IDNs, procedures can have zero or more parameters and an optional return type. Each parameter has a "direction" and a type. In RPC IDL the allowed directions are `in` and `out`. In LIPC IDN the allowed directions are `in`, `out`, and `inout`. The `inout` direction is equivalent to a specification of two parameters, one `in` and one `out`, both of the same type.

RPC IDL limits `out` parameters to be arrays or pointers. This is because RPC IDL views those as being suitable types for assignment. LIPC IDN is not concerned about what is done with a returned (`out`) value. Thus LIPC IDN does not restrict the type of an `out` parameter. In practice, the following correspondence holds between parameter types:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RPC parameter type	LIPC parameter type
T *	T
T [...]	array (integer range (...)) of (T)

This correspondence is not one-to-one, a knowledge of the application semantics will be needed to select the best match.

LIPC IDN requires that a procedure declaration list all the termination conditions (exceptions, errors) that might occur as part of the semantics of the procedure. RPC IDL does not. See Termination Declarations (D.2.4).

RPC IDL has three RPC specific attributes applicable to procedures: `idempotent`, `broadcast`, and `maybe`.

D.2.4 Termination Declarations

In LIPC, a procedure can return normally or in one of a set of named terminations. On normal return, new `out` and `inout` parameter values are provided by the procedure as well as any explicit return value. On a named termination, a different set of values is provided: those declared in the termination declaration.

An LIPC IDN termination declaration consists of a name for the termination, and a list of zero or more types for the returned values.

RPC IDL does not provide any means for declaring non-normal return conditions.

D.3 Primitive Datatypes

D.3.1 Boolean

The LIPC IDN `boolean` type and the RPC IDL `boolean` type are identical.

D.3.2 State

The LIPC IDN `state` type generator is similar to the RPC IDL `enum` type generator, except that the values of a `state` datatype are unordered.

In translating a RPC `enum` datatype into LIPC IDN, a knowledge of the application semantics will be needed to select the best match (`state` versus `enumerated`).

D.3.3 Enumerated

The LIPC IDN `enumerated` type generator and the RPC IDL `enum` type generator are identical.

In translating a RPC `enum` datatype into LIPC IDN, a knowledge of the application semantics will be needed to select the best match (`state` versus `enumerated`).

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.3.4 Character

The LIPC IDN `character(R)` datatype consists of all characters in standard character repertoire R.

There are four RPC IDL character datatypes: `char`, `ISO_LATIN_1`, `ISO_UCS`, and `ISO_MULTI_LINGUAL`.

The RPC IDL `char` datatype appears to be implementation defined, but is guaranteed to contain at least the ISO 646 character repertoire. Thus, if `char` is used only in a portable manner, we have

RPC type	LIPC equivalent
char	character (iso standard 646)
ISO_LATIN_1	character (iso(1) standard(0) 8859 part(1))
ISO_UCS	character (iso(1) standard(0) 10646)
ISO_MULTI_LINGUAL	character (iso(1) standard(0) 10646 multi-lingual-plane)

The three RPC IDL "ISO" character types imply a representation as well as repertoire.

D.3.5 Ordinal

The LIPC IDN `ordinal` type is similar to `integer range (0..)`. The closest RPC IDL analog would be `unsigned hyper`.

D.3.6 Time

RPC IDL has no analog to the LIPC IDN `time` type generator.

A possible encoding of the LIPC IDN `time(unit,radix,factor)` type into a RPC IDL type would be `hyper` with the interpretation that a `hyper` value of h has the interpretation

$$h * (radix^{-factor}) * unit$$

when viewed as a value of type `time(unit,radix,factor)`.

D.3.7 Integer

LIPC IDN provides a single `integer` type of unbounded range, while RPC IDL provides 8 integer types as follows:

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RPC type	LIPC equivalent
hyper	integer range (-2^{63} .. 2^{63}-1)
long	integer range (-2^{31} .. 2^{31}-1)
short	integer range (-2^{15} .. 2^{15}-1)
small	integer range (-2^{7} .. 2^{7}-1)
unsigned hyper	integer range (0 .. 2^{64}-1)
unsigned long	integer range (0 .. 2^{32}-1)
unsigned short	integer range (0 .. 2^{16}-1)
unsigned small	integer range (0 .. 2^{8}-1)

In translating an LIPC `integer` into RPC IDL, a knowledge of the application semantics will be needed to select the best match. A perfect translation is not possible.

D.3.8 Rational

RPC IDL has no analog to the LIPC IDN `rational` type.

A possible encoding of the LIPC IDN `rational` type into a RPC IDL type would be

struct { hyper numerator, denominator; }

where the denominator is greater than 0 and the numerator and denominator are coprime.

D.3.9 Scaled

RPC IDL has no analog to the LIPC IDN `scaled` type generator.

A possible encoding of the LIPC IDN `scaled(radix,factor)` type into a RPC IDL type would be `hyper` with the interpretation that a `hyper` value of h has the interpretation

h * ($radix^{-factor}$)

when viewed as a value of type `scaled(radix,factor)`.

D.3.10 Real

LIPC IDN provides a `real` type generator, with the granularity of the approximation given as ($radix^{-factor}$).

RPC IDL provides two approximations to real numbers: `float` and `double`. The precision and range of these two types are unspecified, although they may be intended to be similar to the IEC 559 single and double precision types. If this is so, the LIPC analogs of these types are

RPC type	LIPC analog
float	real (2,24)
double	real (2,53)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.3.11 Complex

RPC IDL has no analog to the LIPC IDN `complex` type generator.

A possible encoding of the LIPC IDN `complex(radix,factor)` type into a RPC IDL type would be one of

```
struct { float real_part, imaginary_part; }
struct { double real_part, imaginary_part; }
```

where the choice of `float` or `double` would be based on which type was a better approximation to `real(radix,factor)`.

D.3.12 Void

LIPC IDN and RPC IDL provide an identical `void` type. However, the use of `void` in RPC IDL is restricted to (1) the target of a pointer, or (2) the return type of a procedure.

A pointer to `void` would seem to have little use. However, in RPC IDL there seems to be an implicit assumption (perhaps inherited from C) that any `T*` can be coerced to and from a `void*` without loss of information. Thus, `void*` can be used to pass pointer data opaquely through an interface. The LIPC IDN analog of this is `private`.

D.4 Type Qualifiers

RPC IDL has no facilities corresponding to the six type qualifiers defined by LIPC IDN: Range, Selecting, Excluding, Extended, Size and Subtype.

D.5 Generated Datatypes

D.5.1 Choice

The LIPC IDN `choice` type generator provides a pairing of a tag value and a value of one of a number of alternative element types. The tag value determines which of the alternative elements types is used.

The RPC IDL `union` type generator has the same semantics as the LIPC IDN `choice` type generator, except that (in RPC IDL) the type of the tag is restricted to be an integer type, `boolean`, `char`, or an enumeration type. In LIPC IDN the tag type may be any exact datatype. An exact datatype is one that is not `real`, not `complex`, and not generated from `real` or `complex`.

Both LIPC IDN and RPC IDL allow the tag portion of a `choice` or `union` to be omitted from the value. This can be done only when the `choice` or `union` is embedded in a larger datatype (or parameter list) one of whose fields provides the tag value.

RPC IDL prohibits the use of a conformant or conformant-varying array as a `union` alternative. LIPC IDN has no such restrictions.

Note that the RPC IDL syntax allows for new named `union` types.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.5.2 Pointer

The LIPC IDN `pointer to (T)` type generator and RPC IDL `T*` type generator have basically the same semantics. However, RPC IDL provides two attributes to modify the meaning of its pointers, and significantly restricts the use of pointers.

RPC type	LIPC equivalent
[ptr] T *	pointer to (T)
[ref] T *	pointer to (T) excluding (null)

The RPC IDL `[ref]` attribute also provides information to an RPC service, including an assertion that there is no aliasing involving the data referenced by the pointer during RPC invocations.

RPC IDL considers certain pointers to be equivalent to arrays or as substitutes for arrays. Thus in translating RPC IDL pointer types into LIPC IDN, a knowledge of the application semantics and the detailed rules of RPC IDL will be needed to select the best match (`pointer` versus `array`).

See the Type Declaration (D.2.1) and Procedure clauses for further comments on pointers.

D.5.3 Procedure

The RPC IDL does not support procedure types. Procedures in LIPC are defined as operations in RPC (see D.2.3).

D.6 Aggregate Datatypes

D.6.1 Record

The LIPC IDN `record` type generator and the RPC IDL `struct` type generator are identical.

Note that the RPC IDL syntax allows for new named `struct` types.

D.6.2 Set

RPC IDL does not provide a `set` type generator. The LIPC IDN concept of `set` would presumably be modelled as a RPC IDL array. For example,

LIPC type	RPC analog
set of (T)	struct { long size; T element [size]; }

Thus, in translating a RPC IDL `array` type into LIPC IDN, a knowledge of the application semantics is necessary to determine if what is really needed is a `set`.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.6.3 Bag

RPC IDL does not provide a bag type generator. The LIPC IDN concept of bag would presumably be modelled as a RPC IDL array. For example,

LIPC type	RPC analog
bag of (T)	struct { long size; T element [size]; }

Thus, in translating a RPC IDL array type into LIPC IDN, a knowledge of the application semantics is necessary to determine if what is really needed is a bag.

D.6.4 Sequence

RPC IDL does not provide a sequence type generator. The LIPC IDN concept of sequence would presumably be modelled as a RPC IDL array. For example,

LIPC type	RPC analog
sequence of (T)	struct { long size; T element [size]; }

Thus, in translating a RPC IDL array type into LIPC IDN, a knowledge of the application semantics is necessary to determine if what is really needed is a sequence.

D.6.5 Array

The LIPC IDN `array (I) of (T)` type generator defines indexed collections of values. A particular LIPC array value associates an element value in `T` to each index value in `I`. Thus, all values of a particular LIPC array type have the same "length," which is determined by the size of the index type `I`. The index type can be any finite type.

The RPC IDL array type generator defines indexed collections of values as well. However, the only permitted index type is a range of integers.

RPC type	LIPC equivalent
T [a..b]	array (integer range (a..b)) of (T)

Both LIPC IDN and RPC IDL permit arrays to have multiple dimensions (index types).

Both LIPC IDN and RPC IDL provides a means of allowing array bounds to be determined by dependent values. See the clause on Dependent Values (D.9) below.

RPC IDL makes a distinction between the storage bounds of an array and the valid-data bounds of the same array. LIPC IDN specifications are independent of this sort of representation information.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

A RPC IDL conformant array is one that uses dependent values in specifying the storage bounds of an array. A RPC IDL varying array is one that uses dependent values in specifying the valid-data bounds of an array.

Since RPC IDL does not distinguish sequences, bags, and tables from arrays, some uses of RPC arrays may be more properly viewed as sequences. See the clauses on Sequences (D.6.4), Bags (D.6.3), and Tables (D.6.6).

D.6.6 Table

RPC IDL does not provide a `table` type generator. The LIPC IDN concept of `table` would presumably be modelled as a RPC IDL array of structs. For example,

LIPC type	RPC analog
table (FL)	struct { long size; struct {FL'} row [size]; }

where FL' is the RPC equivalent of the LIPC field list FL.

Thus, in translating a RPC IDL `array` type into LIPC IDN, a knowledge of the application semantics is necessary to determine if what is really needed is a `table`.

D.7 Derived Datatypes and Generators

D.7.1 Naturalnumber

The RPC IDL analog to the `naturalnumber` type would be `unsigned hyper`.

D.7.2 Modulo

The RPC IDL analog to the `modulo(n)` type would be an `unsigned` integer type large enough to represent the values 0 through n-1.

D.7.3 Bit

The RPC IDL analog to the `bit` type would be `boolean`.

D.7.4 Bitstring

The RPC IDL analog to the `bitstring` type would be

```
struct { long size; T element [size]; }
```

If the `bitstring` were of fixed size, the best RPC IDL analog would be a `boolean` array.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.7.5 Characterstring

The LIPC IDN `characterstring(R)` datatype contains strings (sequences) of characters of type `character(R)`. Thus strings over any character set can be defined.

RPC IDL can defined strings of chars, bytes, or structures containing only bytes. These latter forms are used to handle character repertoires other than ISO 646 at a representation level rather than a logical level.

RPC type	LIPC equivalent
[string] char *	characterstring (ISO-646)

In translating string types from LIPC IDN to RPC IDL the logical concept of a character repertoire must be replaced by a specific representation in terms of bytes. This requires selecting a particular representation.

In the reverse translation, knowledge of the application semantics will be needed to determine the character repertoire intended.

D.7.6 Timeinterval

RPC IDL has no analog to the LIPC IDN `timeinterval` type generator.

A possible encoding of the LIPC IDN `timeinterval(unit,radix,factor)` type into a RPC IDL type would be `hyper` with the interpretation that a `hyper` value of h has the interpretation

$$h * (radix^{-factor}) * unit$$

when viewed as a value of type `timeinterval(unit,radix,factor)`.

D.7.7 Octet

The LIPC IDN `octet` type is identical to the RPC IDL `unsigned small` type.

D.7.8 Octetstring

The RPC IDL analog to the `octetstring` type would be

```
struct { long size; unsigned small element [size]; }
```

If the `octetstring` were of fixed size, the best RPC IDL analog would be an `unsigned small` array.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

D.7.9 Private

The LIPC IDN `private(n)` datatype contains opaque data n bits long.

The RPC IDL `byte` datatype contains opaque data, presumably 8 bits long. Thus,

RPC type	LIPC equivalent
byte	private (8)

Some uses of `byte` may correspond better to the LIPC `octet` type (which is not opaque).

D.8 Other RPC Datatypes

The RPC datatypes `handle_t`, `pipe T`, and `context_handle` are specific to the RPC service, and are opaque. They have no direct LIPC equivalent.

Opaque service specific types are best handled abstractly as new private types.

The RPC datatypes `error_status_t` is specific to the RPC service, however it is not opaque.

RPC type	LIPC equivalent
error_status_t	integer range (0 .. 2^{32}-1)

D.9 "Dependent Values"

A "dependent value" is an identifier, used in the specification of the type of a field (of a record) or a parameter (of a procedure), which is not defined to be a constant. Rather it is the name of a field in an enclosing record, or the name of a parameter in the same procedure parameter list.

D.10 Cross References

The above discussion has been organized around the LIPC datatypes. The following table lists each RPC datatype and the clause above that discusses it.

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale

RPC datatype	Discussed in
boolean	Boolean (D.3.1)
byte	Private, (D.7.9), Octet (D.7.7)
char	Character (D.3.4)
ISO_LATIN_1	Character (D.3.4)
ISO_UCS	Character (D.3.4)
ISO_MULTI_LINGUAL	Character (D.3.4)
small	Integer (D.3.7)
short	Integer (D.3.7)
long	Integer (D.3.7)
hyper	Integer (D.3.7)
unsigned small	Integer (D.3.7)
unsigned short	Integer (D.3.7)
unsigned long	Integer (D.3.7)
unsigned hyper	Integer (D.3.7)
float	Real (D.3.10)
double	Real (D.3.10)
pointer	Pointer (D.5.2)
array	Array, (D.6.5), Sequence (D.6.4), Set (D.6.2), Bag (D.6.3), Table (D.6.6)
string	Characterstring (D.7.5)
enum	Enumerated, (D.3.3), State (D.3.2)
struct	Record (D.6.1)
union	Choice (D.5.1)
procedure	Procedure (D.5.3)
void	Void (D.3.12)
context_handle	Other RPC Datatypes (D.8)
handle	Other RPC Datatypes (D.8)
pipe	Other RPC Datatypes (D.8)
error status	Other RPC Datatypes (D.8), Termination Declarations (D.2.4)

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS
Not for Resale

ICS 35.060

Descriptors: data processing, information interchange, computer software, programming languages, computer interfaces, models.

Price based on 69 pages

Copyright International Organization for Standardization
Provided by IHS under license with ISO
No reproduction or networking permitted without license from IHS Not for Resale