Information Technology - Language-Independent Procedure Calling Working Draft 6.0 Project number: JTC1.22.16

Document Number ISO/IEC JTC1/SC22/WG11 N344

November 24, 1992

Ken Edwards (Project Editor)
IBM Corporation
555 Bailey Avenue
P.O. Box 49023
San Jose, CA 95161-9023
Phone: (408) 463-3095
FAX: (408) 463-3114
e-mail: edwardsk@stlvm7.vnet.ibm.com

Willem Wakker (SC22/WG11 Convenor)
ACE Associated Computer Experts bv
van Eeghenstraat 100
1071 GL Amsterdam
The Netherlands
Phone: +31 20 6646416
FAX: +31 20 6750389
e-mail: willemw@ace.nl

Contents

0. Introduction

The purpose of this draft International Standard is to provide a common model for language standards for the concept of procedure calling. The Language-Independent Procedure Calling standard 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, called procedures would run 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 called 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 the Language-Independent Procedure Calling standard 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 the 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 draft International Standard.

The Language-Independent Procedure Calling standard is a specification of a common model for procedure calling. This international standard 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 this international standard 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 the LIPC is intended for use by languages so that they may provide standard mappings from their native procedure model. The LIPC will rely on the Language-Independent Datatypes standard for the definition of datatypes that are to be supported in the model for procedure calls provided by the LIPC.

1. Scope

This draft 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 draft International Standard will include such features as procedure invocation, parameter passing, completion status, and environmental issues relating to non-local references and state.

The model for procedure calls that is specified in the LIPC is intended to be used by the Remote Procedure Call standard as the base model for remote calls with extensions being applied by RPC where they are necessary to support RPC specific features of procedure calling. The Interface Definition Notation contained in the LIPC is intended to be shared between the LIPC and the RPC standards with the RPC standard applying appropriate extensions to support remoteness.

This 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 an argument.

2. References

ISO CD 11404: Language-Independent Datatypes, Working Draft #6.1

ISO CD 11578-1.2: OSI RPC Specification - Part 1: Model

ISO CD 11578-2.2: OSI RPC Specification - Part 2: Interface Definition Notation

ISO 8824-ISO 8825: Abstract Syntax Notation - One

3. Definitions

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

3.1 **actual parameter**: A value that replaces a formal parameter during a particular procedure call.

3.2 **argument**: A value communicated between a caller and called procedure via a procedure call.

3.3 **Association**: Any mapping from a set of symbols to values.

3.4 **ASN.1**: Abstract Syntax Notation - One

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

3.6 **called procedure**: The procedure which is invoked by a procedure call.

3.7 **caller**: A sequence of instructions which invokes another procedure.

3.8 **client interface binding**: The possession by the caller of an interface reference.

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

3.10 **execution sequence**: 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.

3.11 **formal parameter**: Identification of a parameter in the definition of a procedure.

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

3.13 **implementation defined**: Possibly differing between processors, but defined for any particular processor.

3.14 implementation dependent: Possibly differing between processors, and not necessarily defined for any particular processor.

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

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

3.17 interface type: A collection of procedure types, and a mapping of a set of names to this collection of types.

3.18 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.

3.19 interface closure: A collection of procedure closures, and a mapping of a set of names to this collection of closures.

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

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

3.22 interface type: A collection of named procedure types.

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

3.24 IDN: Interface Definition Notation

3.25 LID: Language-Independent Datatypes

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

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

3.28 procedure call: The act of invoking a procedure.

3.29 procedure closure: A pair <image, association> where the association defines the mapping for the image's global symbols and no others. Procedure closures are the values of procedure type referred to in the LID.

3.30 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.

3.31 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.

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

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

3.34 procedure return: The act of return from the called procedure with a specific termination.

3.35 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.

3.36 procedure value: A closed sequence of instructions that is entered from, and returns control to, an external source. Within the text of this draft International Standard, the use of the term procedure refers to procedure value.

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

3.38 RPC: Remote Procedure Call

3.39 symbol: A reference in a program text to a value or a box holding a value.

3.40 termination: One of several predefined responses to a procedure call.

3.41 unmarshalling: The process of disassembling the transferred parameters, possibly converting them, for use by the called procedure on invocation or by the caller upon procedure return.

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

4. Definitional conventions

4.1 Formal Syntax

This draft 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 representation. In this clause, the word mark 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 draft 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 valid sequences 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.

4.2 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 draft 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.

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.

5. Compliance

An information processing entity may comply with this draft International Standard by mapping the native calling mechanism of the entity to the model of procedures that is defined in the LIPC.

> **Note:** The general term "information processing entity" is used in this clause to include anything which processes information and contains the concept of procedure calling. Information processing entities for which compliance with this draft International Standard may be appropriate include other standards (e.g., standards for programming languages or language related facilities), specifications, and common procedure libraries.

5.1 Modes of conformance

A information processing entity claiming conformance to this draft International Standard shall conform in either or both of the following ways:

1. It shall allow programs written in its language to call procedures written in another language and supported by another processor, using the model of procedure calls as provided by

clauses 5.1.1, 6, 7, 8, 9, 10. In this case it is said to conform in (and be capable of operating in) client mode.

2. It shall allow programs written in another language to call procedures in its language (i.e. it will accept and execute procedure calls generated by another processor which is executing a program in that other language and which is operating in client mode, and return control to that client processor upon completion), using the model of procedure calls as provided by clauses 5.1.2, 6, 7, 8, 9, 10. In this case it is said to conform in (and be capable of operating in) server mode.

> **Note:** It is also possible in principle for a client processor to use the model for procedure calls defined in this draft International Standard to call procedures in the same language; running 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.

5.1.1 Client mode conformance

In order to conform in client mode, a language processor shall define a mapping from its own language procedure calling mechanism to the common language-independent procedure calling mechanism (LIPC) defined in this draft International Standard.

> **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.

5.1.2 Server mode conformance

In order to conform in server mode, a language processor shall define a mapping from the model of procedure calls defined in the LIPC to its own procedure call model.

> **Note:** 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.

6. Model

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 LID. Value will also contain boxes, procedure closures, and other kinds of values as described below.

6.2 Box

A box is a model of a variable or container that holds a value of a particular type. Boxes exist and are manipulated at runtime. They many 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:

```
create:              --> Value
write:  Box x Value  -->
read:   Box          --> Value
```

Create finds a new box, never before used. Write associates a new value with a given box. 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 not stipulated.

> **Note:** The type Box is different from any datatype introduced in LID.

Boxes imply the existence of a global state, which is the set of all existing boxes and their currently assigned values.

All three Box operations take the current global state as an implicit input value Create and read produce a new current state as an implicit output value.

> **Note:** The global state exists as a modelling concept only. No individual program, running 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. All boxes are members of the set Value.

6.3 Symbol

A symbol is a reference in a program text to a value (or a box holding a value). These are the values that the program can directly manipulate during execution. A particular program's symbols fall into three disjoint categories:

- Global symbols are used to refer to values (including boxes) that exist prior to invocation.
- Local symbols are used to refer to boxes that are created at invocation (the local "stack frame" variables).
- Argument symbols are used to refer to values (including boxes) that are the arguments of a particular invocation.

6.4 Procedure image

A procedure image is the embodiment of a sequence of program instructions. Inherent in a procedure image is the list of global, local, and argument symbols used within the image and the procedure type which it embodies. (The term "procedure type" is defined in LID.) There are four operations defined on procedure images:

```
gsyms: Image --> Sequence(Symbol)

lsyms: Image --> Sequence(Symbol)

asyms: Image --> Sequence(Symbol)

spec:  Image --> Procedure_Type
```

Gsyms returns the global symbols of the image. Lsyms, asyms, and spec return (respectively) the local symbols, argument symbols, and the procedure type.

Procedure images are typically constructed as part of compilation, according to the rules of the particular programming language involved.

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 an association that maps x to y

A + B     denotes an association that satisfies

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

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

rng (A)   denotes the set of values { A(x) | x is in dom(A) }
```

6.6 Procedure closure

A procedure closure is a pair <image,association> where the association defines the mapping for the image's global symbols and no others. In particular, the local and argument symbols have no mappings. Procedure closures are the values of procedure type referred to in the LID.

A procedure closure, some of whose global symbols are not mapped, is said to be a partial procedure closure.

> **Note:** An example of a partial procedure closure is the value of a procedure A nested within a procedure B before procedure B is activated. This is partial because references from A to B's local variables cannot be mapped until the activation of B.

Procedure closures are typically constructed as part of compilation, or during execution, according to the rules of the particular programming language involved. All procedure closures are members of type Value.

6.7 Basic procedure invocation

A basic procedure invocation is an invocation that does not involve translating arguments and return values. Basic invocation is described by the following operation:

```
invoke: Procedure_Closure x Sequence(Value) --> Status x Sequence(Value)
```

where Status is the set of termination identifiers (see LID). The first sequence of values represents the input arguments to the invocation. The second sequence of values represents the values returned by the invocation. The status represents the termination condition, including the "normal" termination.

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

```
Let <A1,...An> = asyms(I)
    <L1,...Lm> = lsyms(I)

For i = 1 to m, do

    LBi = create()

Let Q = A + [A1 -> V1]  + ... + [An -> Vn]
          + [L1 -> LB1] + ... + [Lm -> LBm]

Then

    "Run the image I in the context of association Q"
```

Running an image in a context is a primitive notion defined by the programming language processor (or standard) for the language in which I is written. When, and if, this process completes, the result will be

1. a number of changes to the boxes which comprise the global state, and
2. a value in Status x Sequence(Value).

The association Q is lost at this point. The boxes created in forming Q are no longer accessible unless the programming language permits them to be "returned" in some fashion. (See the section on Associates below.)

6.8 Extending LID

The datatypes in LID are all "static" types. No notion of time or state is needed to understand them. However, the datatype Box introduces both the notion of time and state. In this "dynamic" world model, the LID concepts need to be extended. We need to address "values that change", and procedures that change them.

Consider a simple aggregate like a record. Various programming languages can handle static record values, records that can be changed element by element, and records that can be changed as a whole. Defining separate LID-style types for all combinations of various aggregates with various levels of mutability would cause a combinatorial explosion of types.

Thus, the LID aggregates should not be changed, but mutability should be added as an orthogonal concept. The Box datatype is exactly what is needed. New datatypes can be defined by combining boxes and other LID types.

The concept of procedure closure, introduced above, is the appropriate extension of the LID concept of procedure value to the dynamic world model.

6.9 Instances of values

In a static world, there is no need for a distinction between various instances of a value. Two instances cannot be distinguished unless

1. one instance can be changed without effecting the other (impossible in a static model),

or

2. the instances are at a different location (and location is not a property of LID values).

The first notion of "instance" can be modelled by putting values in boxes. The second notion can be handled as well by extending boxes to have locations. The LIPC does not attempt to model location. However, adding location to boxes is recognized as a compatible extension of the LIPC model.

6.10 Pointers

A box value x has an "identity", x, which can be passed to procedures, and assigned to variables. It also has a value, distinct from the identity x, which can be changed using write, and accessed using read.

The LID Pointer datatype is operationally quite similar to the Box datatype. The only distinction is that the value pointed to by an LID pointer cannot be altered. Thus, an LID pointer is an immutable Box, and Box is the appropriate semantics for Pointer in a dynamic model.

For simplicity, LIPC identifies Box as the underlying semantics of Pointer. The type Pointer(T) denotes any box that is constrained by usage to only hold values of type T.

6.11 Interface closure

An interface closure is a collection of named procedure closures. More precisely, it is an association that maps a set of symbols (names) to procedure closures.

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. Recall that dom(I) = {Sue, Mary, Sam}. Thus, dom(I) is the set of procedure names in the interface closure I. The procedure closure in I named by Mary is denoted I(Mary).

6.12 Interface type

An interface type is a collection of named procedure types. More precisely, it is an association that maps a set of symbols (names) to procedure types.

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

```
IT = [Sue --> XT] + [Mary --> YT] + [Sam --> ZT]
```

is the interface type corresponding to the interface closure I introduced in the preceding section.

6.13 Specifications

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

```
spec: Image --> Procedure_Type
```

spec can be generalized to procedure closures by

spec: Procedure_Closure --> Procedure_Type

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

and spec can be further generalized to interface closures by

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)]

6.14 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 arguments 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 IN and INOUT arguments (in order), and r_1 through r_{rn} are the OUT and INOUT arguments (in order) plus the explicit "returns" result (if present).

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

n = an (the number of arguments is correct)

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

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

< status, <W_1,...W_m> >

If C is written in a programming language that preserves type correctness, 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",

m = rn, and

W_j is a value of type RT_j (for all j in 1..m)

If 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)

6.15 Associates

A value x is an associate of another value y if x can be "extracted" in some fashion from y. This section defines a number of associate functions which can be used to talk about the "accessibility" of values.

The first such function is Immediate Associates:

```
IAssoc: Value --> Set(Value)
```

If x is a value of some non-aggregate type defined in LID, then IAssoc(x) is then empty set. If x is a value of some aggregate type defined in LID, 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) = {v}.

If x is a procedure closure,

IAssoc(x) = {}.

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).

Intuitively, Assoc(x) consists of all values that can be extracted from x by applying various extraction operations on aggregates and read 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

$rng(A)$ union $\{V_1,...V_n\}$

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

$Z = Assoc\ (\ rng(A)$ union $\{V_1,...V_n\}\ )$

The invocation of $< I,A >$ on $< V_1,...V_n >$ can potentially write any box in Z. It can also return any value in Z, and build new aggregates out of such values.

Note: It can also create new boxes.

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

Note: For example, "own variables" of other procedures are not included in Z.

To include such indirectly accessible values, define an Generalized Immediate Associates function:

```
GIAssoc: Value --> Value
```

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

GIAssoc(x) = rng(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

x is in GAssoc(x)

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

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 (rng(A) union $\{V_1,...V_n\}$)

With the help of other procedure closures, this invocation can potentially write any box in GZ. It can also return any value in GZ, build new aggregates out of such values, and return them as well. If y is a value returned by the invocation of $<I,A>$ on $<V_1,...V_n>$, then the values in GAssoc(y) that existed before the invocation must all be in GZ.

Note: It is assumed that for a procedure closure $<I,A>$ to invoke another procedure closure $<J,B>$, $<J,B>$ must be:

1. in rng(A) (the most common case)
2. accessible from an input argument,
3. constructed out of accessible values.

6.16 Argument translations

When a procedure invocation is required to cross between execution contexts, it may not be possible to pass the argument 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 caller'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.

Argument 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, LIPC must be able to model them.

Argument translations can be modelled in the following way. Let C be an arbitrary procedure closure, and let TF and TB be procedure closures that do forward and backward argument 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 arguments
2. invokes C with the translated arguments, 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:** It doesn't 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 than calling C1.

A more precise definition of wrap would be:

```
wrap: Procedure_Closure x Procedure_Closure x 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 argument 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:** TF and TB are the only places where Value (the union of all types) is used in a conceptual context.

** EXAMPLES **

Example (1):

Let's assume that we want to model a remote procedure call (RPC) from client address space (CAS) to 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.

6.17 Defining Translation Procedures

Translation procedures typically need to take a complex value V and replace only certain portions of V, leaving the rest of V "congruent" to the original. For example, 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).

Let T be some mapping from values to values:

```
T: Value --> Value
```

T is an identity on datatype Q if for all values v of type 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 arguments as F. T maps F to F′ if for all values $v_1,...v_n$ in the input domain of F,

$$T(F(v_1...v_n)) = F'(T(v_1), ... T(v_n))$$

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.

TF is defined as a translation procedure that replaces all the 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) intersect Boxes

2. For i = 1 to n,

   C_i = create()

3. Return Z(V), where Z: Value --> Value is a mapping satisfying

   For any box B_i, $Z(B_i) = C_i$

   Z preserves all aggregate types except Box

   Z is an identity on all non-aggregate types

6.18 Execution Context

An execution context is the instance of the objects satisfying the external references necessary to allow the procedure to operate, whose lifetime exceeds the lifetime of the procedure call, that can be referenced by the instantiated procedure. An execution context can contain one or more invocation contexts, although any particular procedure has associated with it a specific execution and invocation context. Each invocation context defines the scope of language semantics, such as local names. Data utilized by a specific invocation context can be either local or remote. Local data is defined to be data which has a scope that is limited to the invocation context of the procedure. External data is defined to be data which has a scope that spans multiple invocation contexts within a specific execution context.

> **Note:** External data does not necessarily have to be accessible to every procedure defined in a particular execution context.

6.19 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 language-independent 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 caller and the procedure being called is referred to as the called procedure.

> **Note:** It is possible for a called procedure to also be a caller if it makes a call to another procedure in order to complete its desired function.

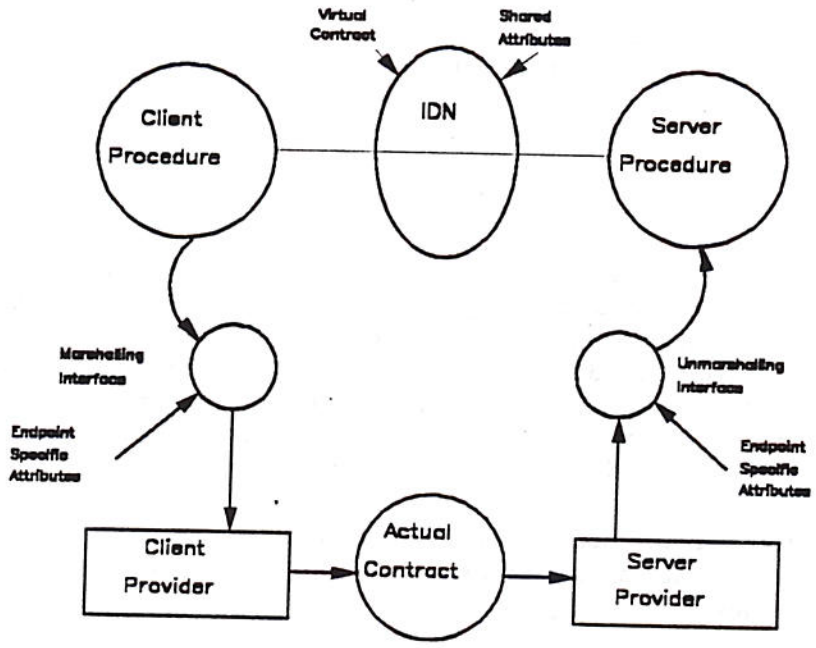
Procedures have the ability to exchange data between the client and server via the use of parameters (see 6.4). In addition, client and server procedures may also share data through the use of global data (see 6.4.6). In order for the parameters specified by the caller to be interpreted correctly, the parameters are required to be marshalled (see 6.4.7) to a base form for transmission that is shared by both the client and the server procedure. After the data has been transmitted, the called procedure must then unmarshall (see 6.4.7) the data from the base form into datatypes

that are defined in the server language or language binding to the LID for that particular language.

> **Note:** An example of the process of marshalling and unmarshalling of parameters would be if a Pascal caller made a call to a Fortran called procedure passing a single character parameter by value. The Pascal "char" type would map to a LID character. In order to have the LID character be transmitted to the called procedure, the LID character is marshalled to an ASN.1 "char" form, for example, which is a form that would be understood by both the client and server procedures. The ASN.1 "char" 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:

Language–Independent Procedure Call Model



This model illustration how the client and called procedures communicate when their implementations conform to this draft International Standard. The virtual contract between the caller and called procedure is defined by the Interface Definition Notation contained within this draft 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 transmittable 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 called 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.

7 Interface Definition Notation

The Interface Definition Notation is the means defined in this draft International Standard for specifying declarations for procedures, procedure parameters, datatypes, and attributes. This concrete notation supports the datatypes defined in the Language-Independent Datatypes standard. For a language processor to comply with this draft International Standard, a binding from the procedure calling mechanism on that language processor to the IDN defined in this draft International Standard needs to be specified in a binding standard. Included in this binding standard should be the inward and outward mappings for the language's datatypes to the types defined in the LID.

7.1 IDN Grammar Syntax

7.1.1 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 ";"}.
```

> **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*.

7.1.1.1 Type references

If an *identifier* is used in an interface type definition to refer to an *type-spec*, it is called a *type-reference* (see 7.1.9).

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-spec* 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-spec* 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*.

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-spec* of this *type-decl*.

7.1.1.2 Value References

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

a) a *value-expr* 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-expr* or *enumeration-identifier*. Otherwise, it is determined at the time of procedure invocation or termination.

7.1.2 Import Declaration

```
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.

7.1.3 Declarations

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

7.1.4 Value Declarations

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

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.
```

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

All *value-identifier*s shall be unique within the immediately containing *interface-type*.

An interface shall only define constants of type "integer", "real", "character", "boolean", or "enumerated".

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

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-expr* declared in another *value-decl* or to an enumeration literal (if the specified type is an enumeration).

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

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

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.

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

state-literal = identifier.

ordinal-literal = digit {digit}.

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

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

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

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

fraction = "." digit{digit}.

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

```
real-part = real-literal.

imaginary-part = real-literal.

void-literal = "nil".
```

7.1.5 Datatype Declarations

```
type-decl = "type" type-identifier "=" type-spec | 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-spec* may be used in the interface (e.g., to define the type of a parameter in a procedure declaration). The syntax and semantics of the *parameterized-type-decl* is given in clause 6.2.2.8.

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

The semantics of all datatypes given in this document are consistent with ISO CD 11404 Common Language Independent Datatypes.

```
type-spec = 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-spec*. The *type-identifier* defined in the immediately containing *type-decl* is a synonym for the *type-spec* 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.

7.1.5.1 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.
```

7.1.5.1.1 The integer datatype

```
integer-type = "integer" [subtype-spec].
```

7.1.5.1.2 The real datatype

```
real-type = "real" [range]
            ["relative_error" relative-error].
```

```
relative-error = real-literal.
```

7.1.5.1.3 The character datatype

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

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

repertoire = object-identifier.
```

The value of a *repertoire* shall be an object identifier specified in ISO 10646 or ISO 7350, or an object identifier obtained by the procedures specified in ISO 2375, to identify collection of characters.

7.1.5.1.4 The boolean datatype

```
boolean-type = "boolean".
```

7.1.5.1.5 The enumerated datatype

```
enumerated-type = "enumerated" "("enumeration-identifier
                      {"," enumeration-identifier}")"
                      [subtype-spec].

enumeration-identifier = identifier.
```

An enumerated type is ordered; an *enumeration-identifier* is considered less than any other *enumeration-identifier* if it appears textually earlier than it in the list of *enumeration-identifiers* in the *enumerated-type.*

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

7.1.5.1.6 The octet datatype

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

According to the LID, the octet type is the derived type: array (1..8) of (bit).

7.1.5.1.7 The procedure datatype

```
procedure-type = "procedure" "("parameter-decls")"
                     ["returns" "("return-arg")"]
                     [termination-list].
```

A *procedure-type* is used to define a reference to a closure. One can invoke such a reference just as a procedure is invoked. The declaration of a *procedure-type* defines the type and direction of its parameters. For a function it defines the type of its return value. It may define a list of terminations (exceptional outcomes). Each termination given in the *termination-list* shall refer to a termination defined in a *termination-decl.*

7.1.5.1.8 The state datatype

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

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

state-value = state-literal | parametric-value.

parametric-value = identifier.
```

7.1.5.1.9 The ordinal datatype

```
ordinal-type = "ordinal".
```

7.1.5.1.10 The time datatype

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

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

radix = value-expr.

factor = value-expr.
```

7.1.5.1.11 The bit datatype

```
bit-type = "bit".
```

7.1.5.1.12 The rational datatype

```
rational-type = "rational".
```

7.1.5.1.13 The scaled datatype

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

7.1.5.1.14 The complex datatype

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

7.1.5.1.15 The void datatype

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

7.1.5.2 Generated Datatypes

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

7.1.5.2.1 The record datatype

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

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

field = field-name ":" type-spec.

field-name = identifier.
```

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

7.1.5.2.2 The select datatype

```
select-type = "choice" "("discriminants")" "of"
                  "("alternative-field {"," alternative-field}")".

discriminant = value-reference.

alternative-field = subtype-spec field-specifier | default-alternative.

default-alternative = "default" field-specifier.
```

```
field-specifier = field | field-name ":" "void".
```

A select datatype consists of a *value-reference*, called the *discriminant*, and a list of *alternative-fields*. Each *alternative-field* consists of a subrange of the *discriminant* type and either a *field-name* and a *type-spec*, or the void specifier. At most one *alternative-field* may be the *default-alternative*, containing the keyword 'default' in place of the subrange.

The type of the *discriminant* shall be either an *integer-type*, a *character-type*, a *boolean-type*, or an *enumerated-type*. The subranges defined in the *alternative-field*s shall be disjoint and shall be specified in accordance with the rules governing the *discriminant* type.

During procedure invocation and termination, exactly one *alternative-field* is selected for each *select-type* parameter. The value of a *select-type* parameter is of the type specified by the selected *alternative-field*. If the *field-specifier* of the selected *alternative-field* is void, then the *select-type* has no meaningful value. An *alternative-field* is selected if the value of the *discriminant*, at procedure invocation/termination time, lies within the subrange associated with that *alternative-field*. If the value of the *discriminant* does not lie within the subrange of any *alternative-field*, and a *default-alternative* is specified, then the *default-alternative* is selected.

If the *select-type* is an input or input/output value, then the value referenced by the discriminant of the *select-type* shall be an input value. The value of the discriminant at procedure invocation shall be used to determine which *alternative-field* is selected. If the *select-type* is an output value and the discriminant is an input value, then the value of the discriminant at procedure invocation shall be used to determine which *alternative-field* is selected. If the *select-type* is an output value, then the value of the discriminant at procedure termination shall be used to determine which *alternative-field* is selected.

All *field-name*s shall be unique within their immediately containing *record-type* or *select-type*.

7.1.5.2.3 The array datatype

```
array-type = "array" "("array-bounds-list")" "of" "("type-spec")".

array-bounds-list = array-dimension {"," array-dimension}.

array-dimension = lower-bound ".." upper-bound | typed-array-bound.

typed-array-bound = defined-datatype.
```

An array datatype includes a specification of the base type of the array, identified by *type-spec*, and each *array-dimension* of the array. Each *array-dimension* has a *lower-bound* and *upper-bound* of the same type. This type shall be either the integer type or an enumerated type. The *lower-bound* shall be less than the *upper-bound*. If the *typed-array-range* is used to specify the *array-dimension*, the resulting type shall be an enumerated type or an integer subtype with a finite *lower-bound* and *upper-bound*. Array bounds may be constant or may be specified as being determined during procedure invocation and/or termination. Constant array bounds may be specified using either a constant literal or an *identifier* declared in a *value-decl*. Non-constant array bounds may be described using a *value-reference* that refers to a *field* of a *record-type*, a formal *parameter* of a procedure or termination, or a *return-arg* of a procedure. In this case the bounds are determined at procedure invocation or termination as dictated below.

If an in or inout array has an *array-bound* specified by a *value-reference*, then that *value-reference* shall be an in parameter, and the value of the *array-bound* is the value of that *value-reference* at the time of the call.

If an out array has an *array-bound* specified by a *value-reference*, then that *value-reference* may be either an in, inout, or out parameter. When the *value-reference* is an in parameter, the value of the *array-bound* is the value of that *value-reference* at the time of the call. When the variable is an out or inout parameter, the value of the bound is the value of that *value-reference* at the time of the return from the call.

If an array with one or more variable bounds is contained in a *record-type*, then the array shall be the last *field* in the *field-list*. A record containing an array with variable bounds shall appear only as the last *field* in the *field-list* of another record.

7.1.5.2.4 The pointer datatype

```
pointer-type = [pointer-attribute] "pointer" "to" "("type-spec")".
pointer-attribute = "restricted" | "unaliased".
```

Abstractly, a pointer consists of a tuple, <label, data>. One can view the data as the value of an instance of a datatype, and the label as the abstract address of the data. Whenever two pointers have equivalent labels, they also have the same data. In this case, the pointers are said to be aliased. Two pointers may, however, have the same data but have different labels. In addition, the null label is a distinguished label that is not associated with any data.

Aliasing of pointers takes two forms: static and dynamic. Static aliasing occurs when two pointers have equivalent labels at the same time, e.g., at procedure invocation or at procedure termination. Dynamic aliasing occurs when two pointers have equivalent labels at different times, e.g., when an out pointer has the same label at procedure termination as an in pointer had at procedure invocation.

When a pointer is passed to a called procedure, representations of both the label and the value of the data are passed. Likewise, when a pointer is returned from a called procedure, representation of both the label and the most recent value of the data are returned.

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 called procedure.

A pointer with the 'unaliased' attribute is a pointer that may have the null label, but is neither statically nor dynamically aliased with any other pointer. Unaliased pointers also can be supported efficiently; however, due to the optimized protocol it is impossible to determine whether the label of an inout unaliased pointer with a non-null label was changed to a different non-null label as a result of executing the called procedure.

> **Note:** If a specific restricted pointer has the null label or is aliased with other pointers, or if a specific unaliased pointer is aliased with other pointers, the results are implementation defined. Whether it is assumed that the labels of inout restricted and unaliased pointers never change or always change as a result of executing a called procedure is implementation defined.

A pointer without the 'restricted' and 'unaliased' attributed is a full pointer. Such a pointer may have the null label and may be statically or dynamically aliased with any other full pointer, whether in, out or inout. Consequently, it is possible to determine whether two in full pointers are aliased at the time of procedure invocation; whether two out full pointers are aliased at the time of procedure termination; whether the label of an inout full pointer was changed as a result of executing the called procedure and, if so, whether the returned label was originally passed to

or was created by the called procedure; and whether the label of an out full pointer was originally passed to or was created by the called procedure.

The detection and maintenance of static aliasing of full pointers is performed on each procedure invocation and each procedure termination.

The period during which dynamic aliasing of full pointers is detected and maintained is limited to the duration of a single, non-callback procedure call. Thus dynamic aliasing is detected and maintained on invocation and termination of a non-callback procedure, and all procedure callbacks which occur during its execution. The detection and maintenance of dynamic aliasing does not occur between any two non-callback procedures.

> **Note:** The existence of pointer aliasing in the application that is external to the pointer declarations specified in an interface type definition may result in pointer inconsistencies or undefined behavior either during the execution of a called procedure, its callbacks, or in the calling procedure after the called procedure terminates.

7.1.5.3 Subtypes

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

select-element = value-expr | range.

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

lower-bound = value-expr.

upper-bound = value-expr.
```

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

7.1.6 Procedure Declarations

```
procedure-decl = [residence-indicator] "procedure" procedure-identifier
                 "("[parameter-decls]")"
                 ["returns" "("return-arg")"]
                 [termination-list].

residence-indicator = "client" | "server".

procedure-identifier = identifier.

parameter-decls = param-decl {"," param-decl}.

param-decl = direction parameter.

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

parameter = parameter-name ":" type-spec.

parameter-name = identifier.

return-arg = [identifier ":"] type-spec.

termination-list = "raises" "("termination-reference
                       {"," termination-reference} ")".
```

A *procedure-decl* declares the signature of a procedure supported by the interface. Procedures may be identified as residing in the client, making them possible targets for procedure callbacks

from the server. If a procedure is not identified as residing in the client then it shall reside in the server. In addition to the procedure name, the procedure declaration defines the type of its parameters. For functions, it defines the type of its return value. The procedure declaration may define a list of terminations (exceptional outcomes). Each termination given in the *termination-list* shall refer to a termination defined in a *termination-decl.*

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

A *procedure-decl* also defines the *direction* of each parameter; i.e., whether the parameter is an input, an input/output, or an output parameter.

All *parameter-names* shall be unique within the immediately containing *procedure-decl*, *procedure-type* or *termination-decl.*

If an *identifier* is specified for a *return-arg*, then that *identifier* shall be distinct from all *parameter-names* in the immediately containing *procedure-decl* or *procedure-type.*

7.1.7 Termination Declarations

```
termination-decl = "termination" termination-identifier
                   "("[parameter {"," parameter} ]")".

termination-identifier = identifier.
```

A *termination-decl* declares the parameters associated with a specific termination and associates a *termination-identifier* with this termination. If a specific procedure includes this *termination-identifier* in its *termination-list*, then this termination becomes a valid termination of that procedure.

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

7.1.8 Parameterized Types

```
parameterized-type-decl = "type" type-identifier
                              "("formal-value-parms")" "=" type-spec.

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

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

value-parm-type-spec = type-spec.
```

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-spec.* Each *formal-value-parm* is itself an *identifier* that can be referenced from within the *type-spec.* References to these *formal-value-parms* can only be used in place of *value-exprs* within the *type-spec* (e.g., in place of an array bound).

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

The *type-identifier* introduced by a *parameterized-type-decl* can be used anywhere a *type-spec* 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.*

7.1.9 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-spec* of a *parameterized-type-decl.* An *actual-value-parm* must be supplied for each *formal-value-parm* of the *parameterized-type-decl.* The type of an *actual-value-parm* must be the same as the type of the corresponding *formal-value-parm.* The semantics of the resulting *type-spec* is that obtained by replacing each *formal-value-parm* reference within the *type-spec* by the corresponding *actual-value-parm.*

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

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

identifier = letter {pseudo-letter}.

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 = "_".
```

7.1.9.1 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.*

7.1.9.2 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.

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 7.1.9.1.

7.1.9.3 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-spec* 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.*

7.1.9.4 Value references to value-exprs

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-expr* 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-expr* 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-expr* of this *value-decl*.

7.1.9.5 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*.

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-expr*, 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-expr*, 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-expr* in the definition denoted by the *interface-synonym*, then the *value-reference* refers to the matching *enumeration-identifier*.

7.1.9.6 Termination References

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

7.1.10 User Defined Letters

The set of letters in the character set defined by a processor shall be user specified. The default set of letters defined by the LIPC are the upper case letters 'A' through 'Z'. A user defined set of letters shall include the default set of letters.

> **Note:** User defined letters allow an implementation to have internationalized procedure-ids.

> **Note:** Issues concerning case sensitivity are the responsibility of the link-editor which is outside the scope of the LIPC standard.

8 Parameter Passing

Any datatype defined in the Common Language-Independent Data Types standard can be the datatype of a formal parameter of a language-independent procedure call. The LIPC defines parameter passing solely on the passing of values. Therefore an actual parameter may be any expression yielding a value of the datatype required by the call. The parameter passing model defined in this draft International Standard is a strongly typed model.

Note: Weak typing can be accomplished by relaxing association rules and adding implicit type conversions in the language bindings to this International Standard.

There are four basic types 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 as Specified
4. Call by Value Returned when Available

8.1 Call by Value Sent on Initiation

This is the simplest form of parameter passing. The formal parameter of the called procedure requires 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 called 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.

8.2 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 called procedure. The evaluation and passing of the actual parameter takes place if and only if the called procedure requests it.

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

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

Note: An example would be the current date and time.

Call by Value Sent on Request could be regarded as a call of an implicit procedure parameter where the called procedure does the evaluation one time. Any further reference in the called procedure to the formal parameter simply uses the value supplied. The called procedure does not issue a further request for a value.

8.3 Call by Value Returned on Termination

In this type of parameter passing, the virtual contract is that at the termination of the call, the called 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:** This type of parameter passing is better know 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.

> **Note:** In a closely coupled environment where providing the actual destination (hardware address) to the called 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 called procedure, which then sends its returned value directly there. This is an additional service level function that the called procedure contracts to perform for the caller, which does not affect the logical division of responsibility at the virtual contract level.

> **Note:** 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 as Specified accommodates function procedures through the use of an additional anonymous parameter.

8.4 Call by Value Returned when Available

In this type of parameter passing, the called 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 termination of the call, or some time later. What time is chosen is determined by the binding of the LIPC based service and is not a matter for the LIPC itself. All the LIPC model requires is that this possibility be accommodated for. The virtual contract is that whenever the called 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.

8.5 LIPC Parameter Passing related to Common terminology

The following notes map the common parameter passing mechanisms that exist languages to the four defined parameter passing schemes that are defined in this draft International Standard.

> **Note:** (1) Call by Value (In parameters): This is the simplest of all common parameter passing mechanism and appears directly in the LIPC as Call by Value Sent on Initiation (see clause 6.4.1). The virtual contract is fulfilled by the client evaluating the actual parameter and sending the value to the called procedure, and the called procedure accepting it. No further action is required of the caller. 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.

> **Note:** (2) Call by Value Return (Out parameters): This common parameter passing mechanism is also directly supported in the 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 called procedure contracts to set. The called 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 allows that distinction to be made at the language binding service contract level without disturbing the virtual contract model.

Note: (3) Call by Value Send 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 Return as Specified (see clause 6.4.3). 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 clause 6.4.1). Equivalently, it can be expanded into two implicit parameters being 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 called 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 called procedure and sends it to that destination.

Where the language binding or service contract passes the destination itself to the called procedure as part of the copy-in/copy-out, the called 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" and any attempt by the server procedure to violate that condition is an exception.

Note: (4) 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.

This is not passed as an in/out parameter due to the fact that 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 called 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 potential significant overhead, which may result in Call by Reference not being supported in such services.

8.6 Global data

Global data refers to data that is 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 the Language-Independent Procedure Calling standard shall 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 recommended that implementations choose to 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.

Note: 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.

Note: 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).

Note: 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.

8.7 Parameter Marshalling / Unmarshalling

It is necessary that data to be communicated between the caller and the called procedure be assimilated into a transmissible form. This transmissible form will allow the client and called 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 standard.

Note: An example of a form that would be suitable for this use would be Abstract Syntax Notation - One.

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

8.8 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 called 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.

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-returned). An access path via pointer parameters implies access to all lower levels, including the primitive datatype values referenced by the lowest level pointers.

8.9 Private types

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

Note: A private type can be considered as an octet stream that can have no operations performed on it.

9 Run-time Control

9.1 Terminations

An implementation conforming to the Language-Independent Procedure Calling standard shall provide a method for raising and handling terminations that occur during the initialization, execution, or termination of a procedure call. Raising a termination does not necessarily imply that the called procedure should be terminated immediately, however terminations that are raised should not be ignored all together by the implementation. Some examples of possible terminations include:

- hardware or software detected events which may or may not be critical to the proper execution of the application
- asynchronous events
- cancellation of a call
- invalid interface instance identifier
- normal termination of a procedure call which returns the output and input/output parameters
- aborting a procedure
- cancelling a procedure

9.1.1 Cancelling a procedure

To cancel a procedure call means to issue a command from outside that 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 caller. 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 procedure being called. In both cases, the caller receives a notification via an implementation defined termination raising mechanism.

9.1.2 Abnormal Termination

A procedure terminating 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 successfully complete the function(s) requested by the caller. The caller is notified by the implementation defined terminations 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 case one.

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. One can 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 caller, the only difference between the two cases is that in the first case, the error specified is one of a predefined set specified in the International Standard (see 9.1.4). In the second case, it is an application-specific condition code specified in some other, perhaps application-specific, standard.

9.1.3 Normal termination

A procedure terminating 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.

9.1.4 Predefined conditions

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

- called procedure unavailable, call not executed
- client or called procedure does not have defined mapping to IDN
- value out of range for parameter datatype
- cancellation of call
- insufficient resources available to complete call
- normal termination of call

10 Execution Control

10.1 Synchronous and Asynchronous Calling

The issue of whether of not a call executes synchronously or asynchronously is outside the scope of the LIPC standard. The LIPC does not inhibit 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.

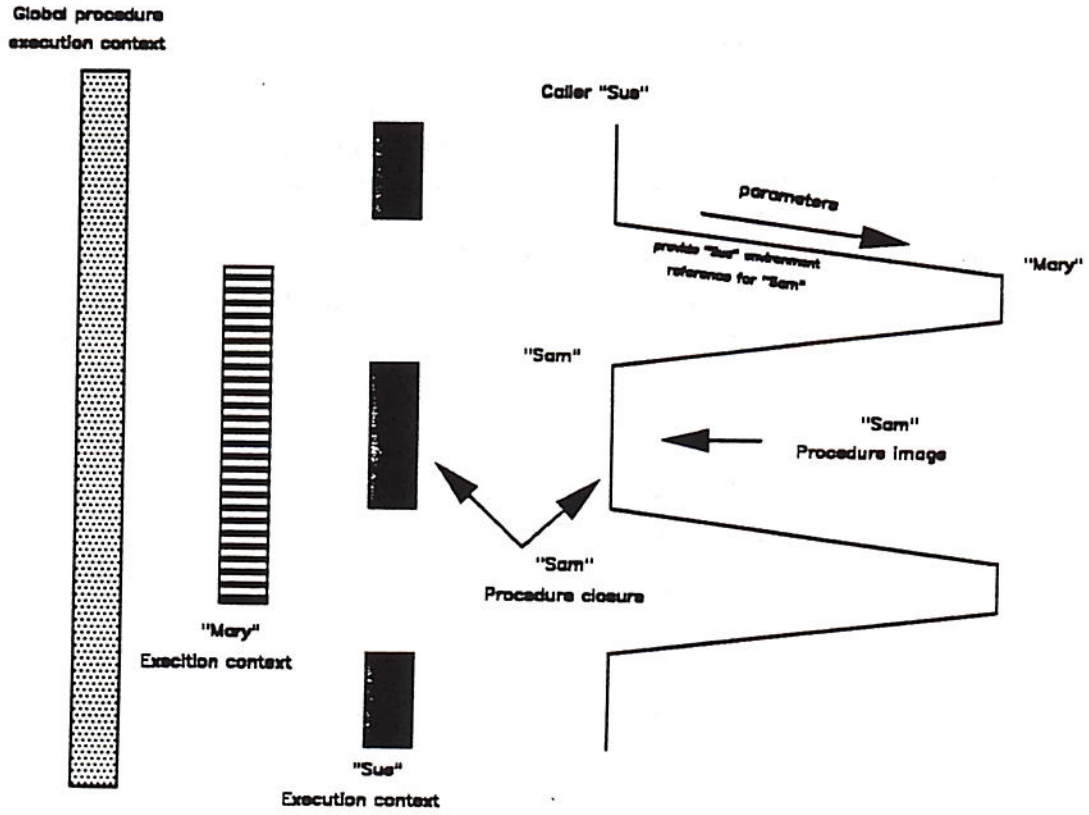
10.2 Recursion

The Language-Independent Procedure Calling standard does not prohibit recursion. It is outside the scope of the LIPC as to how an implementation should implement a recursive procedure call.

> **Note:** Implementors should be aware that optimization considerations for LIPC calls needs to take recursion into account.

Appendix A - Model diagram (alternative)

This appendix reflects some of the concepts relating to clause 6 of this draft International Standard. This model is not complete and requires expansion. It may also be necessary to eventually merge an enhanced version of the diagram with the diagram in clause 6.19.



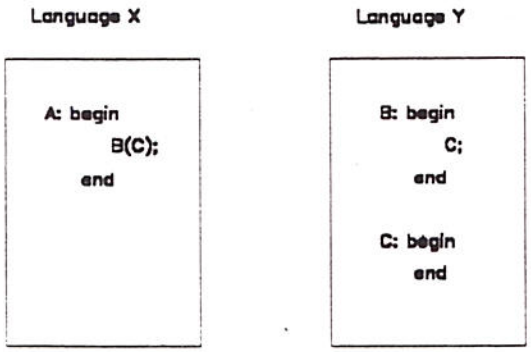
Appendix B - Procedure Parameters

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

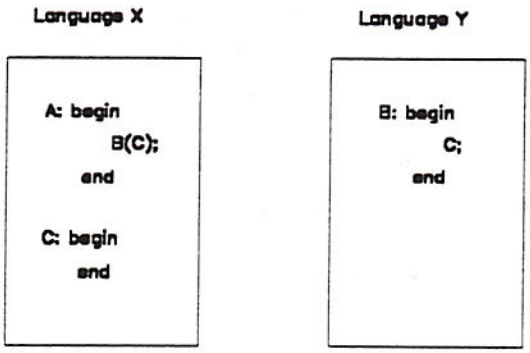
In this case, procedure A in language X calls procedure B in language Y and passes to procedure B a pointer to procedure C which is also in language Y. There shall exist a way for language X to reference procedure C in order to generate a pointer to pass to procedure B. This reference to C shall be referred 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.



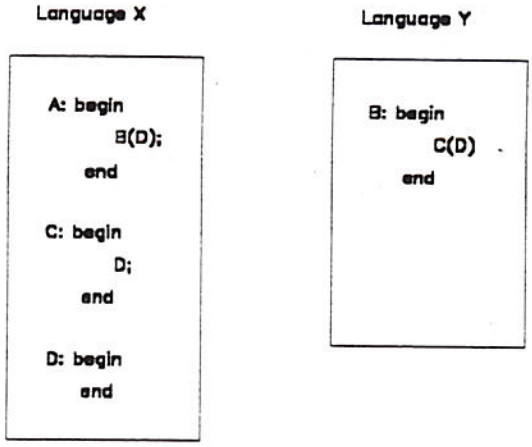
B.2 LIPC Reference / LIPC Access

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.



B.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.



Appendix C - Interface Definition Notation syntax

In this appendix, certain production rules are highlighted as being contained within the Global IDN. This significance of this distinction is that the LIPC has made restrictions on certain rules or has eliminated production rules for use in this draft International Standard.

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

***Global  IDN***********************************************************
* interface-type = interface [interface-synonym ":"]        *
*                  interface-identifier "begin" interface-body "end". *
**************************************************************************

interface-synonym = identifier.

interface-identifier = object-identifier.

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

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

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

import-symbol = identifier.

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

value-decl = "value" value-identifier ":" constant-type-spec "="
                 value-expr.

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-expr = value-reference | integer-literal | real-literal |
                 character-literal | boolean-literal | state-literal |
                 ordinal-literal | time-literal | bit-literal |
                 rational-literal | scaled-literal | complex-literal |
                 void-literal.

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

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

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.

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

state-literal = identifier.

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

```
time-literal = digit{digit} ["."digit{digit}].
bit-literal = "0" | "1".
rational-literal = [-] digit{digit} ["/" digit{digit}].
scaled-literal = [-] digit{digit} [fraction].
fraction = "." digit{digit}.
complex-literal = "(" real-part "," imaginary-part ")".
real-part = real-literal.
imaginary-part = real-literal.
void-literal = "nil".
type-decl = "type" type-identifier "=" type-spec | parameterized-type-decl.
type-identifier = identifier.
type-spec = primitive-datatype | generated-datatype | defined-datatype.
defined-datatype = type-reference [subtype-spec].
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.
***Global  IDN*********************************
* primitive-datatype = integer-type          *
*                    | real-type             *
*                    | character-type        *
*                    | boolean-type          *
*                    | enumerated-type       *
*                    | octet-type            *
*                    | procedure-type        *
*                    | interface-reference   *
*                    | state-type            *
*                    | ordinal-type          *
*                    | time-type             *
*                    | bit-type              *
*                    | rational-type         *
*                    | scaled-type           *
*                    | complex-type          *
*                    | void-type.            *
*********************************************

integer-type = "integer" [subtype-spec].
```

```
real-type = "real" [range]
            ["relative_error" relative-error].

relative-error = real-literal.

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

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

repertoire = object-identifier

boolean-type = "boolean".

enumerated-type = "enumerated" "("enumeration-identifier
                      {"," enumeration-identifier}")"
                      [subtype-spec].

enumeration-identifier = identifier.

octet-type = "octet".

procedure-type = "procedure" "("parameter-decls")"
                     ["returns" "("return-arg")"]
                     [termination-list].

***Global  IDN**********************************************************
* interface-reference = interface-reference [interface-identifier]. *
************************************************************************

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

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

state-value = state-literal | parametric-value.

parametric-value = identifier.

ordinal-type = "ordinal".

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

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

radix = value-expr.

factor = value-expr.

bit-type = "bit".

rational-type = "rational".

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

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

void-type = "void".

generated-datatype = record-type
                     | select-type
                     | array-type
                     | pointer-type.

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

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

field = field-name ":" type-spec.
```

```
field-name = identifier.

select-type = "choice" "("discriminants")" "of"
                "("alternative-field {"," alternative-field}")".

discriminant = value-reference.

alternative-field = subtype-spec field-specifier | default-alternative.

default-alternative = "default" field-specifier.

field-specifier = field | field-name ":" "void".

array-type = "array" "("array-bounds-list")" "of" "("type-spec")".

array-bounds-list = array-dimension {"," array-dimension}.

array-dimension = lower-bound ".." upper-bound | typed-array-bound.

typed-array-bound = defined-datatype.

pointer-type = [pointer-attribute] "pointer" "to" "("type-spec")".

pointer-attribute = "restricted" | "unaliased".

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

select-element = value-expr | range.

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

lower-bound = value-expr.

upper-bound = value-expr.

procedure-decl = [residence-indicator] "procedure" procedure-identifier
              "("[parameter-decls]")"
              ["returns" "("return-arg")"]
              [termination-list].

residence-indicator = "client" | "server".

procedure-identifier = identifier.

parameter-decls = param-decl {"," param-decl}.

param-decl = direction parameter.

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

parameter = parameter-name ":" type-spec.

parameter-name = identifier.

return-arg = [identifier ":"] type-spec.

termination-list = "raises" "("termination-reference
                     {"," termination-reference} ")".

termination-decl = "termination" termination-identifier
                "("[parameter {"," parameter} ]")".

termination-identifier = identifier.

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

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

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

value-parm-type-spec = type-spec.

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

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

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.

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

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

identifier = letter {pseudo-letter}.

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 = "_".
```

Appendix D - Acknowledgements (to be deleted in final version)

This draft International Standard has been developed by ISO/IEC JTC1/SC22/WG11 with assistance from ANSI X3T2. The efforts of the following individuals are acknowledged for their contributions to this project:

- Ed Barkmeyer
- Paul Barnetson
- Jean Bourgain
- Ken Edwards
- Ed Greengrass
- Mark Hamilton
- David Joslin
- Bertrand Leroy
- Brian Meek
- Don Nelson
- Paul Rabin
- Franz Karl Ress
- Craig Schaffert
- Joe Treat
- Willem Wakker