Common Language-Independent Procedure Calling Mechanism
Working Draft 4
(ISO/SC22/WG11 N295)

Document Number ISO/IEC JTC1/SC22/WG11 N295

December 13, 1991

0. Introduction

The purpose of this International Standard is to provide a common reference point to which all languages can relate. The Common Language-Independent Procedure Calling Mechanism is an enabling standard to aid in the development of language-independent tools and services. This International Standards will aid in the development of 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 if not all programming languages include the concepts of procedures and their invocation. The main variance between these methods mainly lies in the ways parameters are passed between the client and server procedures. Procedure calling is a simple concept at the functional level, but at the language level it is not so simplistic. The interaction of procedure calling with datatyping and program structure along with the numerous variations on procedure calling and restrictions on calling that are applied by various programming languages transforms this seemingly simple concept of procedure calling into a much 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 that are evident in the standardized languages let alone those which are not standardized. The existence of the Common Language-Independent Procedure Calling Mechanism does not necessitate 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 CLIPCM from their native procedure calling mechanism, and to be able to accept calls from other programming languages who have defined a mapping to this International Standard.

The Common Language-Independent Procedure Calling Mechanism 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 CLIPCM 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 CLIPCM is intended for use by languages so that they may provide standard mappings from their native procedure model. The CLIPCM will rely on the Common Language-Independent Data Types standard for the definition of datatypes that are to be supported in the model for procedure calls provided by the CLIPCM.

ISO/IEC JTC1/SC22/WG11 reached consensus on this standard on the 1st day of January 199x. At that time, the working group had the following member body participation:

(TBD)

ISO/IEC JTC1 approved this document as an International Standard on the 1st day of January 199x. At that time, the Joint Technical Committee had the following member body participation:

(TBD)

The efforts are acknowledged of all those who contributed to the work of ISO/IEC JTC1/SC22/WG11, and in particular:

(TBD)

The efforts are acknowledged of all those who contributed to the work of X3T2, and in particular:

(TBD)

1. Scope

1.1 This International Standard specifies a model for procedure calls, and a reference syntax for mapping to and from the model. This syntax is referred to as the Interface Definition Notation. The model for procedure calls that is specified in the CLIPCM is intended to be utilized by the Remote Procedure Call standard as the base model for remote calls with extensions being applied by RPC where they are necessary in order to support RPC specific features of procedure calling. The model defined in this International Standard will include such features as procedure invocation, parameter passing, completion status, and environmental issues relating to non-local references and state.

1.2 This standard does not specify:

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

2. References

ISO/SC22/WG11/N233: Common Language-Independent Data Types Working Draft #5.

X3T5/91-124: Proposal for OSI RPC Language-Independent IDN

ISO 8824-ISO 8825: Abstract Syntax Notation - One

3. Definitions

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

3.1 **argument**: Parameter of a procedure.

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

3.3 **by reference**: The passing of an argument such that an effective reference to the actual argument is supplied to the called routine. Modification of the formal parameter does not affect the actual argument value.

3.4 **by value**: The passing of an argument such that only a copy of the actual argument value is supplied to the called routine. Modification of the formal parameter does not affect the actual argument value.

3.5 **by value-return**: A parameter is passed by value-return if the mathematical value of the parameter is made available to the called procedure, but the variable holding that parameter is not made available to the called procedure. The value of the parameter in the called procedure upon completion (which may be different than the initial values) is copied back into the variable that held that parameter in the caller upon exit.

3.6 **call**: The execution of a procedure, starting with the designation of explicit parameters continuing with the modification of the environment including the modification of parameters and concluding with the designation of a return value (if any) to the calling procedure.

3.7 **called procedure**: Procedure which is invoked by a procedure call.

3.8 **calling mechanism**: The logical and functional operations and organization of parameters that define interfaces between communicating procedures.

3.9 **calling procedure**: Procedure which invokes another procedure.

3.10 **CLIDT**: Common Language-Independent Data Types

3.11 **conversion**: (1) Transformation between values that represent the same data item, but belong to a different data type. (2) Transformation between values that represent the same data item but belong to data types in different languages.

3.12 **IDN**: Interface Definition Notation

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**: Data value passed to the called procedure on entry from the calling procedure.

3.16 **input/output parameter**: A pair of related data values representing the transformation of a single parameter during the execution of a procedure. One value is passed to the calling procedure, and the other from the called procedure on return to the calling procedure.

3.17 **interface**: Externally visible characteristics of a set of procedures. These characteristics might include data type declarations, procedure declarations, and exception declarations.

3.18 **marshalling**: Process of collecting parameters, converting them to an standard data representation, and assembling them for transmission.

3.19 **output parameter**: Data value passed from the called procedure on return to the calling procedure.

3.20 **procedure**: An abstraction of an action, command, or operation that can specify implicit and explicit parameters, modify the environment including the implicit and explicit parameters, and possibly provide a return value. Procedures may also be referred to as operations, routines, subroutines, and functions.

3.21 **return**: Upon completion of the finalization process of the called procedure, control is then *returned* to the calling procedure.

3.22 **RPC**: Remote Procedure Call

3.23 **state**: Existence of an environment for a procedure call.

3.24 **unmarshalling**: Process of disassembling a list of parameters from the message in which they were transmitted, and converting them to the format used by the procedure.

4. Definitional conventions

The metalanguage used in this standard to specify the syntax of the language-independent procedure calling mechanism is defined below:

- Brackets [ ] enclose an optional part of the syntax.
- Ellipsis ... indicates that the left clause can be repeated either 0 or more times if it is optional or 1 or more times if it is required.
- Notation punctuation that does not conflict with punctuation characters used in the BNF, appear in a production in the appropriate position. Notation punctuation that does conflict with punctuation characters in the BNF, is enclosed in less-than and greater-than symbols, e.g. <[>. Quotation marks that appear in a production are part of the notation and must appear in the interface definition source.

- Elements in the grammar that are capitalized are terminals of the grammar. For example, < Identifier > is not further expanded. Also, keywords of the notation are terminals of the grammar. For example, the keyword "char" is not further expanded.
- The Interface Definition Notation contains two kinds of keywords. Reserved keywords may not be used as identifiers. Keywords which are not reserved may be used as identifiers, except when used as attributes.
- The punctuation used in the Interface Definition Notation consists of period '.', comma ',', parentheses '(' and ')', slash '/', square brackets '[' and ']', semicolon ';', colon ':', asterisk '*', single quote ''', double quote '"', and equal sign '='.
- White space is a character sequence that can be used to delimit any of the other low level constructs. The syntax of white space consists of a blank, a return, a horizontal tab, a form feed in column 1, a comment, and a sequence of one or more white space constructs.
- A notation keyword, an < Identifier >, or a list of < Digit > s not preceded by a punctuation character must be preceded by whitespace. A notation keyword, an < Identifier > or a list of < Digit > s not followed by a punctuation character must be followed by whitespace. Any punctuation character may be preceded or followed by whitespace.
- The characters "/*" introduce a comment. The contents of a comment are examined only to find the characters "*/" that terminate it. Comments therefore do not nest in the IDN.

5. Compliance

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

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 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 International Standard shall conform in one of three ways:

1. It may allow programs written in its language to call procedures written in another language and supported by another processor, using the model of procedure calls defined in this standard. In this case it is said to conform in (and be capable of operating in) *client mode*.
2. It may 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 defined in this standard. In this case it is said to conform in (and be capable of operating in) *server mode*.
3. It may conform in (and be capable of operating in) both *client mode* and *server mode*.

Note: It is possible in principle for a client processor to use the model for procedure calls defined in this International Standard also 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, an information processing entity shall define a mapping from its own language procedure calling mechanism to the common language-independent procedure calling mechanism (CLIPCM) defined in this International Standard.

Note: If a program using the CLIPCM facility is to be portable between processors for its language which conforms in client mode, then the program and processors will also need to conform to the relevant language standard and the relevant standards binding that language to the CLIPCM and CLIDT standards.

5.1.2 Server mode conformance

In order to conform in server mode, an information processing entity shall define a mapping from the model of procedure calls defined in the CLIPCM to its own procedure call model.

Note: If a procedure is to be portable between processors for its language which conforms in server mode and still to be called by client processors and programs, then the procedure, and the processors, will also need to conform to the relevant language standard and the relevant standards binding that language to the CLIPCM and CLIDT standards.

6. Requirements

6.1 Language-Independent Call Model

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 know as the "client procedure" and the procedure being called is referred to as the "server procedure".

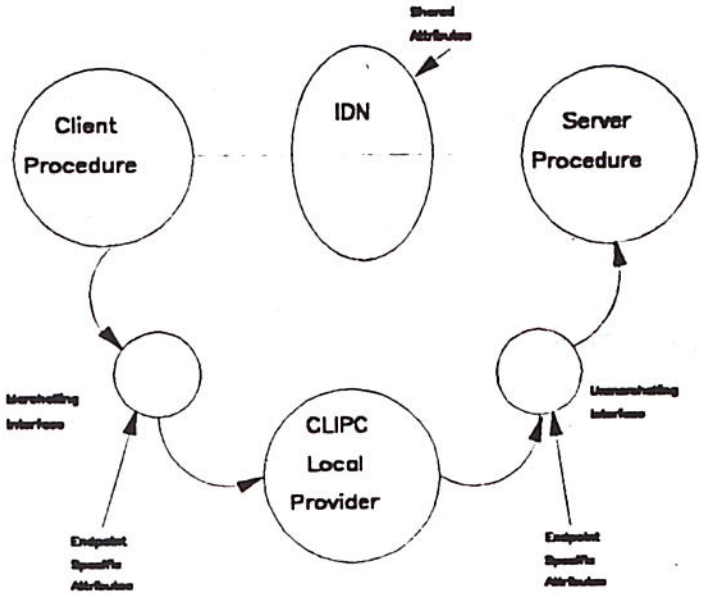
Note: It is possible for a server procedure to also be a client procedure if it makes a call to another procedure in order to complete its desired function.

Procedures have the ability to exchange data between the client and server via the use of parameters (see 6.4). If 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 client procedure 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 server 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 into the CLIDT for that particular language.

Note: An example of the process of marshalling and unmarshalling of parameters would be if a Pascal client procedure made a call to a Fortran server procedure passing a single character parameter. The Pascal "char" type would map to a CLIDT character. In order to have the CLIDT character be transmitted to the server procedure, the CLIDT 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 CLIDT character, which in turn maps to a "character*1" in Fortran.

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

Common Language–Independent Procedure Call Model



This language-independent procedure calling model shall be the base model for the remote procedure call standard. The Interface Definition Notation contained in the CLIPCM is intended to be shared between the CLIPCM and the RPC standards with the RPC standard applying appropriate extensions to support remoteness.

6.1.1 Call Environment

The *call environment* consists of the available resources for the language processor.

Note: An example of one such resource would be a common library.

Within each call environment is an address space which is logically separate for other call environments. The specification of a call that crosses call environment boundaries is out side the scope of this standard.

Note: The Remote Procedure Call standard will define such a specification.

A call environment can contain one or more procedure environments. Each procedure environment defines the scope of language semantics, such as local names. Data utilized by a procedure contained within a procedure environment can be either local or external. Local data is data which has a scope that is limited to a single procedure environment. External data is data which has a scope defined to be the entire call environment. Therefore, each procedure environment contained within a call environment has the ability to access external data that is defined in its call environment.

6.1.1.1 Procedure Environment Initialization/Identification

The procedure environment consists of three basic parts; the body of the procedure, parameters, and an associated state. An implementation conforming to this International Standard shall ensure that the procedure environment is created and initialized in an implementation defined method. Upon creation of a procedure environment, there shall be a unique identifier associated with the newly created environment. The purpose of this identifier is to identify a previously initialized environment that has maintained a certain state from a previous invocation.

The key component of the state of any particular procedure environment is the ability for the procedure environment to maintain values of variables from one instantiation to another. A procedure call that relies on a procedure environment being in a particular state must have access to the unique identifier for that environment; otherwise a new procedure environment would be created which would likely have a state that differs from that of the existing environment.

6.1.1.2 Environment Sharing

As has already been discussed, global data can be shared between procedures whose procedure environments exist in a common call environment. Another form of sharing environments takes place when passing a procedure as a parameter. In order for the called procedure to have the ability to access the procedure it has received as a parameter, it is necessary for the environment identifier to be passed to the called procedure to ensure the appropriate state for the procedure.

6.1.1.3 Procedure Termination

Upon completion of the intended task of the called procedure, the called procedure must terminate its thread of execution and return control back to the thread of execution that initiated the call. Return control back to the calling procedure includes making any necessary value substitutions for parameters passed by value (see clause 6.4.1). In addition to parameter handling, the procedure environment of the calling procedure must be reinstated in an implementation defined manner. In the case where the called procedure must maintain its state for a subsequent call, an implementation conforming to this International Standard shall maintain the state of this procedure environment in an implementation defined manner.

Upon termination of a procedure, the called procedure must communicate the completion status of the call to the calling procedure. Issue resulting from normal and exceptional returns from a called procedure are covered later in this standard under exception handling (see 6.7.1).

6.2 Interface Definition Notation Grammar Syntax

6.2.1 Interface Structure

```
<interface> ::= interface <interface_identification>
                begin <interface_body> end
```

Possible syntaxes for < interface_identification >:

```
<interface_identification> ::= <interface_identifier>
                               [ version <integer_literal> ]

<interface_identification> ::= <interface_identifier>
           [ compatible with <Object_identifier> [, <Object_identifier> ] ...
```

Each interface must have an associated (globally) unique < Object_Identifier >. For convenience, one can also assign a local "name" to an interface that one can locally refer to. Furthermore, this local name can be used within an import list. (See below).

Note: The usage of a local name to identify an interface is for local usage only. When communicating the interface to a nonlocal entity, an < Object_identifier > must be used.

```
<interface_identifier> ::= <Object_identifier>
                         | <local_name> <Object_identifier>

<local_name> ::= <Identifier>
```

List of imported RPC interfaces

```
<interface_body> ::= [<imports>] <declaration>; [ <declaration>; ] ...

<imports> ::= import ( <import_list> )

<import_list> ::= <import> [, <import> ] ...
```

Each import is either identified by a unique Object id or by a local name.

```
<import> ::= <Object_identifier>  | <local_name>

<declaration> ::= <value_decl>
                | <type_decl>
                | <proc_decl>
```

6.2.2 Value Declarations

A value declaration introduces an abbreviation (an Identifier) for a constant value.

```
<value_decl> ::= value  <Identifier> : <type_spec> = <value_expr>
```

For CLIDT, one can construct a constant value for (almost) any type. For RPC, the only value expressions needed are integers, and possibly some others for use within attributes. RPC will therefore restrict the productions in this section.

```
<value_expr> ::= <Identifier>
               | <literal>
               | <qualified_value>
               | <composite_value>
```

An identifier is a reference to a value declaration (or to a parameter if the value expression occurs on the RHS of a parameterized declaration). A literal is a simple immediate value.

```
<literal> ::= <integer_literal>
            | <real_literal>
            | <character_literal>
            | <Boolean_literal>
            | <enumerated_literal>
            | <rational_literal>

<integer_literal> ::= [-]<Digit>...

<real_literal> ::= <integer_literal>.<Digit>...

<character_literal> ::= '<Character>' [ ( <char_set> ) ]
```

String literals are used in building other literals.

```
<string_literal> ::= "<Character>..." [ ( <char_set> ) ]

<Boolean_literal> ::= true | false

<enumerated_literal>  ::= <Identifier>

<rational_literal> ::= <integer_literal>/<Digit>...
```

A qualified value T.V is the value V interpreted as a value of type T. This is used only when there is a unique injection of V into T. For example, Time.″<iso8601-date>″. Not used in RPC, more on this later.

```
<qualified_value> ::= <type_spec> . <value_expr>
```

A composite is used for arrays, lists, tables, and so on. Each such type of composite places additional restrictions on this basic syntax. For example, a record looks like (id: value, id:value, ...). Not used in RPC, more on this later.

```
<composite_value> ::= ( [ elt [, elt] ... ] )

<elt> ::= [ <value_expr> : ] <value_expr>

<integer_value> ::= <integer_literal> | <Identifier>
```

6.2.3 Datatype Declarations

```
<type_decl> :== <new_type_decl> | <type_macro>
```

A < new_type_decl > introduces a new type. This is the same as the current CLIDT "new" keyword. This type is not the same as any other structurally equivalent type. The left hand side can contain free variables that are referenced on the right hand side.

```
<new_type_decl> ::= type <Identifier> [ ( <Identifier> [, <Identifier> ]... ) ]
                        = <type_spec>
```

A < type_macro > introduces an abbreviation -- the left hand side is an abbreviation for the right hand side. Unlike a < type_decl >, a < type_macro > DOES NOT introduce a new type. The left hand side can contain free variables that are referenced on the right hand side.

```
<type_macro> ::=  macro <Identifier> [ ( <Identifier> [, <Identifier> ]... ) ]
                       = <type_spec>

<type_spec> ::= [ <type_attributes> ] <primitive_type_spec>
              | [ <type_attributes> ] <generated_type_spec>
              | [ <type_attributes> ] <type_decl_ref>
              | [ <type_attributes> ] <type_spec> <subtype_qualifier>
```

6.2.3.1 Primitive Datatypes

```
<primitive_type_spec> ::= <integer_type>
                        | <real_type>
                        | <char_type>
                        | <Boolean_type>
                        | <enumerated_type>
                        | <procedure_type>
                        | <octet>
                        | <state>
                        | <ordinal>
                        | <time>
                        | <bit>
                        | <rational>
                        | <scaled>
                        | <complex>
```

RPC restrictions: RPC does not support the types < state >, < ordinal >, < time >, < bit >, < rational >, < scaled >, and < complex >.

Note: The < octet > type given above is not in CLIDT. It can be viewed as the CLIDT type: array [1 .. 8] of bit. For RPC usages, < bit > is almost never useful (whereas < octet > is). Therefore, RPC uses < octet > as the primitive type in place of < bit >.

```
<integer_type> ::= integer

<real_type> ::= real ( <relative_error> )

<relative_error> ::= <integer_value>

<char_type> ::= character [ ( <char_set> ) ]
```

The standard will indicate a default character set to be used if one is not specified.

```
<char_set> ::= <Identifier>

<Boolean_type> ::= Boolean

<enumerated_type> ::= enumerated ( <Identifier> [ , <Identifier> ] ... )

<procedure_type> ::= proc ( <parameter_dcls> )
                     [ returns (<return_arg>) ]
                     [ <exception_list> ]

<octet> ::= octet
```

Note: The IDN does not define context_handles or binding_handles as primitive datatypes. The RPC standard, however, defines them as generated types defined in an standard interface that all interfaces implicitly import:

- type context_handle = array [0 .. c_handleSize] of octet
- type binding_handle = array [0 .. b_handleSize] of octet

```
<state>  ::= state ( <Identifier> [ , <Identifier> ] ... )

<ordinal> ::= ordinal

<time> ::=  time ( <relative_precision> [, <radix>, <factor>] )

<bit>  ::= bit

<rational>  ::= rational

<scaled>  ::= scaled ( <radix> , <factor> )

<complex>  ::= complex ( <relative_error> )
```

6.2.3.2 Generated Datatypes

```
<generated_type_spec> ::= <record_type>
                        | <choice_type>
                        | <array_type>
                        | <ptr_type>
                        | <list>
                        | <set>
                        | <bag>
                        | <table>
```

Note: RPC restrictions: RPC does not support the types <list>, <set>, <bag>, and <table>.

Note: The array type given here differs from the array type in CLIDT in 3 ways: the index set is always integer, the size of the array can vary (like CLIDT lists), and arrays can be multi-dimensional. Except for the last issue, arrays presented here could be viewed as CLIDT lists. However, the last issue is more intricate. As multi-dimensional arrays are viewed as important by RPC, this document presents the more general array concept. Since there is disagreement on this issue, however, the array concept given here is

only preliminary. A joint discussion between CLIDT and RPC committees is needed to iron out these differences.

```
<record_type> ::= record ( <member_list> )

<member_list> ::= <member> [, <member> ] ...

<member> ::=  [ <component_attributes> ] <Identifier>  : <type_spec>

<choice_type>  ::= choice ( <member_list> )
```

The index of each array dimension is implicitly always integer. The array_bounds_list is a sequence of subtypes, one for each array dimension. Each subtype is a range of integer values, where either (or both) sides of the bound are allowed to be "*", indicating an indeterminate bound. An empty range signifies [0 .. *].

```
<array_type> ::= array <array_bounds_list> of <type_spec>

<array_bounds_list> ::= <array_bounds_declarator>
                        [ <array_bounds_declarator> ] ...

<array_bounds_declarator> ::= <[> [ <array_range> ] <]>

<array_range> ::= <array_bound> .. <array_bound>

<array_bound> ::= <integer_value> | *

<ptr_type> ::= pointer_to <type_spec>

<list> ::= list_of <type_spec>

<set>  ::= set_of <type_spec>

<bag>  ::= bag_of <type_spec>

<table> ::= table_of ( <element> , <key> )

<element> ::= <type_spec>

<key> ::= <type_spec>
```

6.2.3.3 Type Declaration References

A <type_decl_ref> is reference to a new type declaration or a type macro. If the reference is only a single identifier, and it occurs on the RHS of a parameterized declaration, it also may be a reference to a parameter.

```
<macro_instance> ::= <Identifier>  [ ( <expr_list> ) ]

<expr_list> ::= <expr> [ , <expr> ] ...

<expr> ::= <type_spec> | <value_expr>
```

6.2.3.4 Subtypes

```
<subtype_spec> ::= <range> | <max> | <min> | <plus> | <restrict> | <view>
```

The range subtype can be used on any ordered type. A precise bound must be a value of that ordered type. The infinities are only usable in subtyping integer and real.

```
<range> ::=  range ( <lower_bound> , <upper_bound> )

<lower_bound> ::= <precise_bound> | neg_infinity

<upper_bound> ::= <precise_bound> | pos_infinity

<precise_bound> ::=  <value_expr>
```

The max and min subtypes can be used to form subtypes of the list, set, bag, and table types. They bound the size of the aggregate.

```
<max>  ::= max ( <integer_value> )

<min>  ::= min ( <integer_value> )
```

Plus is CLID's extended. Restrict is CLID's selected. View is CLID's explicit subtype. These 3 are NOT used in RPC.

```
<plus> ::= plus ( <expr_list> )

<restrict> ::= restricted_to ( <expr_list> )

<view> ::= viewed_as ( <type_spec> )
```

6.2.4 Procedure Declarations

```
<proc_dcl> ::= [ <proc_attributes> ]
               proc <Identifier> ( [ <parameter_dcls> ] )
               [ returns ( <return_arg> ) ]
               [ <exception_list> ]

<parameter_dcls> ::=  <param_dcl> [ , <param_dcl> ] ...

<param_dcl> ::= <direction> <parameter>

<parameter> ::= [ <param_attributes> ] <Identifier> : <type_spec>

<direction> ::= in | out | inout

<return_arg> ::= [ <param_attributes> ] [ <Identifier> : ] <type_spec>

<exception_list>  ::= raises ( <exception_dcl> [, <exception_dcl> ... ] )

<exception_dcl> ::= <Identifier> ( [ <parameter> [, <parameter> ] ... ] )
```

6.2.5 Attributes

```
<type_attributes> ::= <[> <type_attribute> [ , <type_attribute> ] ... <]>

<proc_attributes> ::= <[> <proc_attribute> [ , <proc_attribute> ] ... <]>

<param_attributes> ::= <[> <param_attribute> [ , <param_attribute> ] ... <]>

<component_attributes> ::= <[> <comp_attribute> [ , <comp_attribute> ] ... <]>
```

The rest of this section contains proposed attributes for RPC.

6.2.5.1 Type Attributes

There are two kinds of <type_attributes>.

- (I) endpoint-specific attributes specify how to map the interface type to a particular implementation on either the caller/called side (or both). An endpoint specific attribute contains only local mapping information; therefore, it does not effect any protocol. Examples include: how to map the choice type to discriminated unions.
- (II) Contractual attributes (also called type_restriction_attributes) contain information that both sides must agree to. These attributes may affect protocol. Examples include: type restrictions that indicate a more efficient wire encoding of the datatype (e.g., sparse), a restricted use of pointers (e.g., unaliased) that allows a more efficient marshalling routine to be used, etc.

```
<type_attribute> ::= <type_restriction_attribute>
                   | <endpoint_specific_attribute>
```

The sparse attribute is used in conjunction with the array, list, set, bag, and table types. It indicates that most values of the aggregate are expected to have a default value. The unaliased attribute is used to indicate that a pointer structure contains no aliasing. It can be used only in conjunction with a pointer type.

The nonnull attribute is used to indicate that a pointer cannot have a null value. It can be used only in conjunction with a pointer type. This attribute only makes sense if pointers are allowed to have null values, a still unresolved issue.

```
<type_restriction_attribute> ::= sparse ( <value_expr> | <Identifier> )
                                | unaliased
                                | nonnull
                                | <lifetime>
                                | <procedure_attribute>
```

The < lifetime > attribute indicates that a procedure parameter (i.e., a closure) may only have a restricted lifetime. In particular, the environment in which the procedure parameter lives may only exist during the current call (i.e., the procedure parameter is a "callback") or during the duration of the current caller/called binding.

```
<lifetime>  ::= callback | extended_callback
```

An < endpoint_specific_attribute > may be qualified by the keyword "caller"/"called" to indicate that the attribute applies only to one endpoint.

The local_representation(X) attribute is used to indicate that a given type is represented by a local type X. The encode/decode attributes provide the names of local routines to encode and decode the interface type to/from a local representation. The < switch_type > attribute is used to indicate a "switch" type that is used locally to discern what case a choice is in. (An example is given below).

```
<endpoint_specific_attribute> ::= caller ( <endpoint_specific_attribute> )
                                | called ( <endpoint_specific_attribute> )
                                | local_representation( <Identifier> )
                                | encode( <Identifier> )
                                | decode( <Identifier> )
                                | switch_type ( <s_type> )

<s_type> ::= <integer_type>
           | <char_type>
           | <Boolean_type>
           | <enumerated_type>
                 | <Identifier>
```

6.2.5.2 Procedure Attributes

```
<procedure_attribute> ::= at_most_once | idempotent
```

6.2.5.3 Parameter Attributes

Note: The following param attributes are open for discussion. Param attributes indicate a restricted relationship that must exist between the in and out values of an inout parameter. The same_structure attribute is used in conjunction with an inout pointer parameter to indicate that the "topology" of the structure pointed to by the param does not change during the call. The same_size attribute is used in conjunction with an inout open array, list, set, bag, and table parameter. It indicates that the size of the aggregate does not change during the procedure call. The same_case attribute is used in conjunction with an inout choice parameter to indicate that the case of the choice does not change during the procedure call.

The <dynamic_information> attribute is used to inform the marshalling routine where certain runtime information can be found to help in marshalling (see below).

```
<param_attribute>  ::= same_structure
                     | same_size
                     | same_case
                     | <dynamic_information>
```

6.2.5.4 Component Attributes

The <discriminant_value> attribute is used in each arm of a choice in order to identify the particular arm of the choice with a set of values from the <switch_type> domain. It must be used in conjunction with the <switch_type> attribute. The <dynamic_information> attribute is used to inform the marshalling routine where certain runtime information can be found to help in marshalling (see below).

```
<comp_attribute>  ::= <discriminant_value>
                    | <dynamic_information>

<discriminant_value> ::= case ( <d_value> [ , <d_value> ] ... )

<d_value> ::= <value_expr> | default
```

The <discriminant_is> attribute is used to tell the marshalling routine where the discriminating value for a choice is. (If the choice is embedded within a record, then the discriminating value must be another component of the record. If the choice is a parameter of a procedure, then the discriminating value must be another parameter of the procedure.) Similarly, the <bounds_is> attribute is used to tell the marshalling routine where the bounds of a conformant array or a varying array can be found. The caller attribute is used to indicate that a component of a record or a parameter of a procedure call is provided by the caller only for use by the runtime. It is not part of the type and is not to be transferred on the wire. A typical usage of this attribute is in conjunction with the <discriminant_is> and <bounds_is> attributes. Upon return from the call, this component or parameter is reconstructed by the runtime. Similarly, the called procedure attribute indicates that a component or parameter is not part of the type and is not transferred on the wire. Instead, it is constructed by the runtime and provided to the called procedure. A typical usage of this attribute is in conjunction with the <discriminant_is> and <bounds_is> attributes.

```
<dynamic_information> ::= <discriminant_is>
                        | <bounds_is>
                        | caller
                        | called

<discriminant_is> ::= discriminant_is ( <Identifier> )
```

One can now write the following in an interface:

```
declare T = [switch_type(integer)]
            choice([case(5)] a: int,
                   [case(10)] b: real,
                   [case(default)] c: Boolean);

proc foo([caller] x: integer, [discriminant_is(x)] y: T)
```

```
<bounds_is> ::= first_is ( <attr_var_list> )
              | length_is ( <attr_var_list> )
              | min_is ( <attr_var_list> )
              | size_is ( <attr_var_list> )

<attr_var_list> ::= <attr_var_dcl> [ , <attr_var_dcl> ] ...

<attr_var_dcl> ::= [ <attr_var> ]

<attr_var> ::= <Identifier>
```

6.3 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 CLIPCM 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 CLIPCM standard.

6.4 Parameter Passing

Any datatype defined in the Common Language-Independent Data Types standard can be a formal parameter of a language-independent procedure call. The CLIPCM 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 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

6.4.1 Call by Value Sent on Initiation

This is the simplest form of parameter passing. The formal parameter of the server procedure requires a value of the datatype concerned. The virtual contract is that the caller 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.

Note: This type of parameter passing is probably better known as Call by Value.

6.4.2 Call by Value Sent on Request

The virtual contract for this type of parameter passing is that the caller 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: An example would be 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.

6.4.3 Call by Value Returned as Specified

This type of parameter passing is essentially the "out" equivalent to Call by Value Sent on Initiation. The virtual contract is that the called procedure will supply a value of the datatype of the formal parameter and the caller will accept is and act appropriately.

Note: This type of parameter passing is probably better known as Call by Value Return.

For a conventional "out" parameter it means that the destination is evaluated by the caller at the initiation of the call, and then sends the returned value to that destination at the conclusion of the call.

Note: In the case where the expression is simply the name of the variable, the timing of the trivial evaluation is of no significance, but is necessary to specify to accommodate the more general case, and make it clear that this is the usual evaluation of actual parameters on initiation.

It is necessary to specify that it is the caller and not the called procedure which sends the returned value to the destination, in order to accommodate situations, as in RPC, where the caller and called procedures may be decoupled. 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.

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.

6.4.4 Call by Value Returned when Available

This type of parameter passing is primarily here for completeness to cover the case where the caller does not evaluate the destination for an "out" parameter on initiating the call, but delays it until the returned value is actually received. Therefore, the called procedure can determine the timing of the evaluation to be 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 CLIPCM based service and is not a matter for the CLIPCM itself. All the CLIPCM model requires is that this possibility be accommodated for. The virtual contract is that the caller will receive the returned value when the called procedure sends it, and then evaluate the destination and send the value there.

6.4.5 Relation of Conventional Parameter Passing to the CLIPCM

In this section, the common parameter passing mechanisms that exist in current languages will be mapped to the four defined parameter passing schemes that are defined in this International Standard.

6.4.5.1 Call by Value (In parameters)

This is the simplest of all common parameter passing mechanism and appears directly in the CLIPCM as Call by Value Sent on Initiation (see clause 6.4.1). The virtual contract is fulfilled by the caller 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 called procedure does what it likes with the received value, but can make no further demands on the caller with respect to the actual parameter that generated the value.

6.4.5.2 Call by Value Return (Out parameters)

This common parameter passing mechanism is also directly supported in the CLIPCM by Call by Value Returned as Specified. The virtual contract for this mechanism involves the concept of passing 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 can not 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 CLIPCM allows that distinction to be made at the language binding service contract level without disturbing the virtual contract model.

6.4.5.3 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 CLIPCM 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 the current value held in that destination on initiation of the call is retrieved by the caller and relayed to the called procedure. The destination itself is also recorded. In the virtual contract, the caller receives the returned value, the "out part of the in/out, 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 caller explicitly or implicitly marks the destination as "read once only, write once only" and any attempt by the called procedure to violate that condition is an exception.

6.4.5.4 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.

Note: This is not passed as an in/out due to the fact that this would cause an extra level of indirect addressing.

The virtual contract is that the caller 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 caller needing to take no further action until the call is complete. In a loosely-coupled service environment the service contract will involve caller 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.

Note: 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.

6.4.6 Global data

Global data refers to data that is defined in one procedure environment that can be referenced by another procedure executing in a different procedure environment within the same call environment. Implementations conforming to the Common Language-Independent Procedure Calling Mechanism shall support an implementation-defined mechanism for the sharing and 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.

6.4.7 Parameter Marshalling / Unmarshalling

It is necessary that data to be communicated between the client procedure and the server procedure be assimilated into a transmissible form. This transmissible form will allow the client and server procedures to encode their CLIDT 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 calling procedure must do in order to transform its data into a form that is understood by the called procedure. Unmarshalling of data refers to what the called procedure must do in order take the data passed by the calling procedure and transform this into data suitable for the language of the called procedure. Marshalling is not limited to the calling procedure as 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 calling procedure 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, it would be beneficial to implementations 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.

6.4.8 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 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.

6.5 Synchronous and Asynchronous Calling

The issue of whether of not a call executes synchronously or asynchronously is outside the scope of the CLIPCM standard. It is the intent of the CLIPCM not to 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.

6.6 Recursion

It is the intent of the Common Language-Independent Procedure Calling Mechanism not to prohibit recursion. It is outside the scope of the CLIPCM as to how an implementation should implement a recursive procedure call.

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

6.7 Run-time Control

6.7.1 Condition Handling

An implementation conforming to the Common Language-Independent Procedure Calling Mechanism shall provide a method for handling conditions that occur during the initialization, execution, or termination of a procedure call. Conditions can be defined in a number of ways. Some examples of conditions are:

- hardware or software detected events which may or may not be critical to the proper execution of the application
- asynchronous events
- successful or unsuccessful completion of a unit of work

Systems today communicate information about these conditions in various ways, for example, return codes and/or exceptions, and there is little if any commonality in the use, representation, or communication methodology used for these conditions across these systems. Therefore, in order to provide for consistency in multi-lingual callable services, implementations conforming to the Common Language-Independent Procedure Calling Mechanism shall provide an implementation defined data type for the representation of conditions and for the communication of information required to process the consequences of their existence. Called procedures shall utilize this implementation defined data type for conditions to return information as a feedback code.

Note: Advantages resulting from the condition data type:

- A condition handler can be established to process return information from called services. This method would free the programmer from coding 'invoke then check' type of calls. Instead, a centralized location would be used to handle return information.
- As a shared data type among callable services, condition management and message services, it ties together these components.
- A message that can be displayed or logged in a file is associated with each instance of a condition.
- As a feedback code, it can be stored for later processing.

As a minimum, implementations conforming to the Common Language-Independent Procedure Calling Mechanism should report the following conditions during execution.

- Server procedure unavailable, call not executed.
- Client or server procedure does not have defined mapping from its language to the CLIDT form defined by the IDN.
- Value out of range for data type. (e.g. An integer is passed as a parameter from a Pascal program to a COBOL program that is outside of the allowable decimal range for the COBOL data type.)
- Mechanism broken on server procedure side of call.

Implementations conforming to this International Standard shall also support exception raising in an implementation defined manner. Exception raising occurs when the calling procedure decides for some reason that the called procedure should be terminated immediately.

The condition handling mechanism supported by the implementation shall be made available to the Common Language-Independent Procedure Calling Mechanism in an implementation defined method. An implementation shall document all conditions flagged by the implementation during procedure initialization, execution, and termination.

6.8 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 CLIDT type in the IDN.

Note: A private type can basically be considered as a byte stream that can have no operations performed on it.

Appendix A

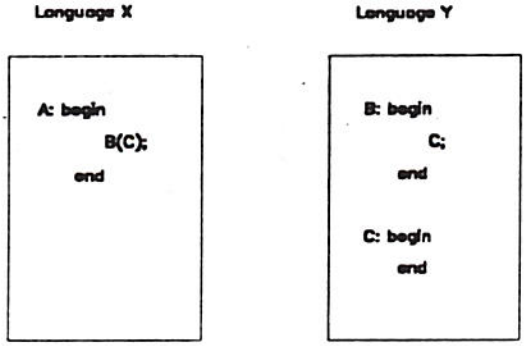
A.1 Procedure parameters

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.1.1 CLI Reference / Local Access

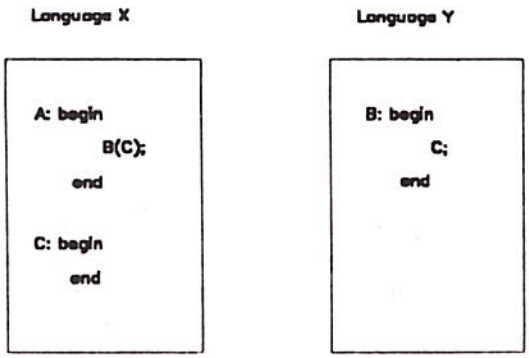
In this case, procedure A in language X calls procedure B in language Y and passes to procedure B a pointer to procedure C which is also in language Y. There shall exist a way for language X to reference procedure C in order to generate a pointer to pass to procedure B. This reference to C shall be referred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

Note: Procedure B must understand how to call procedure C "locally" based on the cli-reference information it was passed.



A.1.2 CLI Reference / CLI 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 *cli-access* since the call crosses the boundary. In addition to this for B to call C, it must have the *cli-reference* of C. This information is obtained from that which was passed from procedure A.



A.1.3 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 *cli-reference*, but it must be in a form that allows the transformation to B's environment and back to its original state.