CapIDL Language Specification[†]

Version 0.1

Jonathan Shapiro, Ph.D. Mark Miller
Systems Research Laboratory
Dept. of Computer Science
Johns Hopkins University

February 13, 2006

Abstract

CapIDL is an interface definition language for capability-based systems. It is loosely derived from the CORBA IDL language, and specialized for the needs of capability-based systems.

This document provides an English-language specification of the CapIDL input language and its intended meaning. CapIDL intentionally does *not* include any specification of transport-level data layout or application-layer serialization rules. It is solely a specification of the interface layer.

Contents

1 Overview

This document is the reference definition for CapIDL. CapIDL is a language for the specification of interfaces between subsystems. It is somewhat based on CORBA IDL. In contrast to CORBA IDL, CapIDL is specialized for interfaces where:

- Interfaces are designated by capabilities. An interface name is therefore a CapIDL type.
- Caller and callee are firmly separated. CapIDL intentionally lacks any means to define interfaces that rely on shared memory or by-reference argument transmission.
- Because accurate reference documentation for interfaces is critical, CapIDL provides mechanisms for

the definition of interface and method documentation, and provides XML as an output language. This XML can be post-processed to HTML, OSDOC, or (presumably) other desired forms. The reference interface documentation for the Coyotos kernel is generated this way.

CapIDL is intended to provide a specification that is neutral with respect to source language, but it attempts to tread the line between being "neutral" and being "naive." As a concrete example of this, consider the definition of the POSIX `read()` function. In languages that allow preallocation of buffer storage, it is desirable to express the fact that the output of the read can be placed directly into this buffer. CORBA provides no means of saying this.

2 Specification Conventions

The primary conventions used in this document concern the presentation of grammar rules. In grammar rule specifications, non-terminals are presented in lowercase italics, categorical terminals are shown in normal face, and literal terminals appear in bold face or within single-quote characters. White space is permitted between tokens. CapIDL is case sensitive; all CapIDL keywords are lowercase.

Except when quoted, the characters { and } indicate metasyntatic grouping following the customary representation of EBNF grammars. The superscript characters *****, +, and **?** indicate, respectively, zero or more, one or more, or an optional item or group. Except when quoted, the character ":" indicates the separation between a non-terminal and its EBNF definition. For categorical terminals, sets of input characters are abbreviated by enclosing them within square brackets. Within such a set, the character '-' denotes an inclusive lexical sequence according to the usual conventions of regular expressions.

3 Theory of Operation

Examples in this section are non-normative.

Conceptually, an interface definition language specifies two things:

1. A collection of (versioned) interfaces, each of which defines constants, types, and methods.
2. A set of name spaces that define containing scopes for these interfaces.

In practice, matters are slightly complicated by pragmatics. It is desirable to be able to specify two (or more) interfaces in separate input files that reside in the same name space, and it is necessary to be able to specify independent name spaces that ultimately can be unified into a single conceptual hierarchical space of identifiers. The CapIDL **package** defines an "open" name space that allows both objectives to be satisfied. The recommended usage pattern follows the Java convention for package names: the defining entity uses a reversed, dotted domain name, as in:

```
package org.coyotos;
```

as its top-level name space.

Logically, packages and the identifiers they contain are defined to occupy a single, unified namespace. It is an (unreliably checked) error for a single identifier `a.b` to refer ambiguously to both a package and a declared element within a package. This restriction is validated by CapIDL when possible. Since interfaces are simultaneously being defined by multiple, non-communicating organizations, it is impossible for a CapIDL implementation to check this requirement in general. In practice, the use of domain names as package names means that most such errors occur within a single, localized group of cooperating developers and are detected.

Coyotos interface definitions may make use of single inheritance. The inheriting interface *extends* its parent interface, providing all methods of the parent plus additional methods unique to the extending interface:

```
interface parent {
  ... declarations ...
};
interface child extends parent {
  ... additional declarations ...
}
```

Interfaces may make use of types defined on other interfaces, provided this does not result in static circular dependency among the interfaces involved. For this purpose, the use of an interface name *as a type* does not create a dependency: an interface name used as a type is conceptually a reference type. In practice, such "lateral" interface cross-references are rare; the more common pattern is for an interface to make general use of other interface names as types, but to rely only on data types defined by itself and its ancestor interfaces.

When used as a type name, an interface name denotes that a capability argument is expected, and that the passed capability is expected to be of the specified interface type (or an extension). The CapIDL design assumes that capabilities are dynamically typed, and does not emit code to type-check capability arguments.

4 Lexical Matters

Character Set The CapIDL input character set is the UTF-8 encoded UNICODE character set.

Comments CapIDL supports two comment formats derived from C and C++. Any sequence beginning with `//` and ending with the next newline is a comment. Any character sequence beginning with `/*` and ending with the next `*/` is a comment. No beginning of comment appearing within a comment is considered lexically significant. For purposes of input tokenization, a comment is considered to be whitespace, and terminates any input token. Comments therefore cannot be used for "token splicing."

Documentation Comments Certain comments have no significance for purposes of programmatic interface specification, but have significance to the humans who use these interfaces. Any comment beginning with `/**` followed by a space or a newline begins a documentation comment that is terminated by the next appearance of `*/` in the input. Similarly, any *consecutive sequence* of comments beginning with `///` and terminating with the next newline is a documentation comment. For this purpose, a "consecutive sequence" of documentation comments is deemed to include any sequence of `///` comments that are separated only by white space.

Many productions in the CapIDL grammar include an underlined `Ident` in a production that defines that identifier. The appearance of this underlined terminal in the indicates that the nearest preceding documentation comment will be associated with the symbol being defined by the present production *provided* that comment has not already been associated with some other symbol.

Identifiers A CapIDL identifier consists of a leading alphabetic character followed by zero or more alphanumeric characters. For this purpose, the underscore character is considered alphabetic:

```
Ident: [a-zA-Z_][a-zA-Z0-9_]*
```

Identifiers beginning with two leading underscores are reserved. In general, a use-occurrence of an identifier may be a "dotted name"

```
dotted_name: Ident {'.' Ident}*
```

With the intended meaning "search for the leftmost identifier according to the usual lexical scoping rules, treat the resolved name as a scope, and search (recursively) within that scope for the succeeding identifiers, but in no case should this search resolve to any identifier definition that lexically follows the use-occurence in any incompletely defined scope in the current unit of compilation.

While the CapIDL input character set is UTF-8 encoded UNICODE, CapIDL identifiers are restricted to the ISO-LATIN-1 subset of the UNICODE identifier specification. In principle, the identifier specification could be extended to UNICODE characters more generally, and we expect that it eventually *will* be extended in this fashion. For the moment, there remain many programming languages whose source code character set is restricted to the ISO-LATIN-1 subset. CapIDL adopts this restriction for the sake of broad compatibility.

Integer Literals An integer literal `IntLit` consists of a sequence of decimal digits:

```
IntLit: [0-9]+
```

Constant Expresions CapIDL allows the use of an arbitrary arithmetic expression using the operators "+", "-", "*", and "/" with their usual meanings and precedence. Parenthesization is permitted. Both unary negation and binary subtraction are supported. Constant expression values are computed using arbitrary precision arithmetic. The associated grammar rules are:

```
const_expr: const_sum_expr
const_sum_expr:
    const_mul_expr
  | const_mul_expr '+' const_mul_expr
  | const_mul_expr '-' const_mul_expr
const_mul_expr:
    const_term
  | const_term '*' const_term
  | const_term '/' const_term
const_term:
    dotted_name
  | literal
  | '(' const_expr ')'
```

with the restriction that a *dotted_name* must reference some previously defined constant. Note that for purposes of expression computation, character literals are considered to denote their corresponding code points as integers.

To Do The specification will soon, but does not yet, specify an input syntax for character, string, and floating point literals.

5 Structure of a Compilation Unit

A CapIDL unit of compilation consists of a package declaration followed by one or more declarations:

```
UOC: package_dcl decl*
package_dcl: package dotted_name
decl:
```

```
    const_dcl ';'
  | typedef_dcl ';'
  | enum_dcl ';'
  | bitset_dcl ';'
  | except_dcl ';'
  | struct_dcl ';'
  | union_dcl ';'
  | namespace_dcl ';'
  | interface_dcl ';'
```

Every declaration defined at package scope is defined within the namespace defined by its containing package. The package and declaration namespace are unified: it is an error for a single identifier `a.b` to refer ambiguously to both a package and a declared element within a package. This requirement is validated by CapIDL when possible, but in the absence of complete input it is not possible for CapIDL to universally detect this error.

6 Types

The types of CapIDL are defined by:

```
type:
    param_type
  | seq_type
  | buf_type
param_type:
    simple_type
  | string_type
  | array_type
  | object
  | dotted_name
```

Where `object` indicates any capability type. Note that certain types may appear within structure, union, and typedef declarations, but may not appear directly as parameter types. This restriction exists to ensure that the CapIDL code generator has a name for certain types that may not be natively expressable in certain target languages.

6.1 Basic Types

The simple types of CapIDL consist of the integer types, floating point types, character types, and `boolean`. The character type `wchar` includes an optional size specifier giving the character code point size in bits. Legal values are 8, 16, and 32. If unadorned the `wchar` type defaults to a 32-bit UNICODE code point.

The `integer` type may carry an optional size qualifier giving the representation size of the integer type in bits. The representable range of the corresponding value is [-2^{n-1},$2^{n-1}-1$]. The types `byte`, `short`, `long`, and `long long` are convenience syntaxes for signed integer types of (respectively) 8, 16, 32, and 64 bits.

The `unsigned` type similarly may carry an optional size qualifier in bits. The representable range of the corresponding value is [0, 2^n-1]. The types `unsigned byte`, `unsigned short`, `unsigned long`, and `unsigned long long` are convenience syntaxes for unsigned integer types of (respectively) 8, 16, 32, and 64 bits.

Note that the unadorned type `integer` indicates an *arbitrary precision* integer value. Such a value has no statically known size. In general, the use of a dynamically sized type requires additional data copying in the stub code generated by CapIDL.

```
simple_type:
    boolean
  | char_type
  | integer_type
  | float_type

char_type:
    char
  | wchar
  | wchar '<' const_expr '>'

integer_type:
    signed_integer_type
  | unsigned_integer_type

signed_integer_type:
    integer
  | integer '<' const_expr '>'
  | byte
  | short
  | long
  | long long

unsigned_integer_type:
    unsigned '<' const_expr '>'
  | unsigned byte
  | unsigned short
  | unsigned long
  | unsigned long long
```

The `float` type may carry an optional size qualifier giving the representation size of the floating point type in bits. All floating point values use IEEE format. Legal values for the floating point size are 32, 64, and 128. The types `float`, `double`, and `long double` are convenience syntax for floating point types of (respectively) 32, 64, and 128 bits.

```
float_type:
    float '<' const_expr '>'
  | float
  | double
  | long double
```

6.2 Enumeration Types

CapIDL supports two enumeration types: enumerations and bit sets. An enumeration declaration takes the form:

```
enum_dcl:
  integer_type enum ident
    '{' enum_def
        { ',' enum_def }* '}' ';'
enum_def: ident
enum_def: ident '=' const_expr
```

In the absence of a provided *const_expr*, the enumeration value of an enumeration element takes on the sequentially next greater value. If the first element of an enumeration does not have a provided *const_expr*, it is assigned the value zero.

The enumeration declaration introduces both new constant definitions and a new type definition.

A bitset takes the form:

```
bitset_dcl:
  unsigned_integer_type bitset ident
    '{' enum_def
        { ',' enum_def }* '}' ';'
enum_def: ident
enum_def: ident '=' const_expr
```

Bitsets are semantically equivalent to enumerations, with the caveat that they may generate different type declarations in some emitted languages. For example, in C++, the following code generates a type error:

```
enum enum_t {
  one = 1,
  two = 2
};
int foo(enum_t arg);
  ...
foo(one|two);
```

If this enumeration is declared in CapIDL using a bitset, the code will be emitted to use a typedef of an unsigned integer type rather than an enumeration, which evades the warning.

6.3 Composite Types

A structure declaration takes the form:

```
struct_dcl:
  struct ident '{' member+ '}' ';'
member:
    struct_dcl ';'
  | union_dcl ';'
  | enum_dcl ';'
  | bitset_dcl ';'
  | namespace_dcl ';'
  | const_dcl ';'
  | element_dcl ';'
element_decl:  type ident
enum_def: ident '=' const_expr
```

A union declaration takes the form:

```
union_dcl:
  union ident '{'
  switch '(' switch_type ident ')' '{'
    case+
  '}' ';' '}'
switch_type:
    integer_type
  | char_type
  | dotted_name
  | boolean
case:
  case_label+ case_def* element_dcl ';'
case_label:
    case const_expr ':'
  | default ':'
case_def:
    struct_dcl ';'
  | union_dcl ';'
  | enum_dcl ';'
  | bitset_dcl ';'
  | namespace_dcl ';'
  | const_dcl ';'
```

Note that structure and union declarations implicitly define a name space.

6.4 Sequence Types

CapIDL provides three classes of sequence types: arrays, sequences, and buffers. An array describes a fixed-length vector of some type, and may appear only in a typedef, structure, or union. Arrays are value types. A sequence describes a dynamically sized (possibly statically bounded) vector of some type. The "content" of a sequence type is generally indirect. A buffer describes a dynamically sized (possibly statically bounded) vector of some type.

The difference between a sequence and a vector lies in their use as parameters. an "out buffer" is actually an "in" parameter describing the size (the limit) of the buffer, and

a compound "out" parameter describing the number of elements actually received (the new length) and the location into which they should be received. Sequences should be used when the destination storage for the out parameter should be allocated by the CapIDL-generated stub procedure. Buffers should be used when the destination storage for the out parameter is preallocated by the caller.

```
array_type:
  array '<' type ',' const_expr '>'
seq_type:
    sequence '<' type '>'
  | sequence '<' type ','
               const_expr '>'
buf_type:
    buffer '<' type '>'
  | buffer '<' type ','
             const_expr '>'
```

6.5 Exceptions

An exception defines an alternate return value for an interface method. Generally, the return value of a method is a union of several exception types plus a "normal" return type. Exceptions *may* have member fields to carry additional information about the exceptional condition. Use of such fields is rare.

```
except_dcl:
    except ident
  | except ident '{' member+ '}' ';'
```

Where *member* is defined above in the discussion of *struct_dcl*.

6.6 Typedef

A new name may be introduced for an existing type using a typedef declaration:

```
typedef_dcl:
  typedef type ident
```

7 Constants

A constant declaration takes the form:

```
const_dcl:
   const simple_type ident
     '=' const_expr ';'
```

A declared constant may be used in any subsequent *const_expr* wherever a *dotted_name* is expected.

8 Name Spaces

A namespace provides a means for wrapping a series of declarations in a common containing scope:

```
namespace_dcl:
  namespace ident '{' decl+ '}'
```

9 Interfaces

All of the preceding exist for the purpose of defining interfaces and their methods. An interface is a (versioned) namespace of methods and other declarations.

```
interface_dcl:
  interface ident
    { extends dotted_name }?
    { raises '(' exceptions ')' }?
  '{' if_def+ '}'
exceptions:
  dotted_name { ',' dotted_name }*
```

Where *if_def* consists of:

```
if_def:
    struct_dcl
  | union_dcl
  | except_dcl
  | namespace_dcl
  | enum_dcl
  | bitset_dcl
  | typedef_dcl
  | const_dcl
  | opr_dcl
```

Note that an interface *raises* clause may refer to exceptions that are declared within the raising interface.

The important production here is the one for *opr_dcl*, which is used to describe methods of interfaces. The general form of *opr_dcl* is:

```
opr_dcl:
    { nostub | client }?
    oneway void ident '(' inparams? ')'
    { raises '(' exceptions ')' }?
  | { nostub | client }?
    ret_type ident '(' params? ')'
    { raises '(' exceptions ')' }?
  | inherits ident
ret_type: void | type
inparams: param_type ident
          { ',' param_type ident }*
params: inout_param_type ident
        { ',' inout_param_type ident }*
inout_param_type: out? param_type
```

A **oneway** method has no expected reply, and must therefore always have a return type of void.

The qualifier **nostub** indicates that the stub procedure has been generated by hand, and CapIDL should not generate a stub for this method.

The qualifier **client** indicates that a method is entirely implemented in a client library, and is not actually part of the interface protocol. Such a method may be included either for descriptive completeness or for backwards compatibility. Note that **client** implies **nostub**.

A method may raise exceptions as indicated in it's *raises* clause. In addition, every method may raise any exception identified for the interface as a whole.

An **inherits** declaration indicates explicitly that an operation is inherited from a parent interface. this declaration has no semantic effect. Its purpose is to allow additional, interface-specific documentation to be added to an inheritor's implementation of a function.

A Change History

This section is an attempt to track the changes to this document by hand. It may not always be accurate!

This section is non-normative.

A.1 Version 0.1

Initial capture.