Shore Data Language Reference Manual[1]

The Shore Project Group
Computer Sciences Department
UW-Madison
Madison, WI
Version 1.1.1

October 27, 1997

[1]This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-92-C-Q508.

Contents

1 Introduction **1**

2 Lexical Syntax **1**

3 Grammatical Conventions **2**

4 Objects and Values **3**

5 Modules **3**
5.1 Examples 4

6 Linking **5**

7 Name Scopes **5**
7.1 Examples 5

8 Interfaces **5**
8.1 Example 6
8.2 More Examples 7

9 Constants **8**
9.1 Examples 9

10 Types **9**
10.1 Examples 11
10.2 External Types 12
10.3 Examples 12

11 Attributes **13**

12 Indexes **13**

13 References and Relationships **14**

14 Operations **15**

15 Appendix A: Collected Grammar **15**

16 Appendix B: Grammar in YACC Syntax **19**

1 Introduction

The Shore Data Language (SDL) is a language for specifying the types of persistent objects. Although the design of SDL is heavily influenced by C++, it is intended to be independent of any particular target language. Definitions written in SDL are compiled by the SDL compiler (see the manual page *sdl(sdl)*) into objects stored in the database. These objects are then used

to generate data declarations appropriate to a particular language, called a *language binding*. (Currently, only a C++ language binding exists).

SDL is derived from and inspired by the object-definition language ODL being developed by the Object Database Management Group (ODMG) as a proposed standard for object-oriented databases. At the time of this writing, the ODMG standard was not suitable for our purposes. That standard actually defines two languages. ODL *per se* is a variant of the Interface Description Language (IDL) defined by the Object Management Group (OMG) in connection with the Common Request Object Broker (CORBA) standardization effort. IDL is essentially a specification language for remote procedure call (RPC) interfaces. As such, it assumes complete encapsulation of objects. For example, it provides no way to specify the public and private parts of an object, since it is assumed that the private parts need not be mentioned in the specification. Thus an IDL (or ODL) specification does not have enough information to determine the layout of the data in an object, or even its total size. As such, it is inadequate for an "object-shipping" implementation in which cached copies of objects are directly manipulated by application programs. Moreover, the ODL specification is currently incomplete. While there is a (nearly) LALR grammar describing the syntax, many of the constructs have no semantics specified.

The ODMG specification also describes a specification language called "ODL–C++". No syntax for this language is given. Rather, it is described informally in terms of modifications of the type-specification fragment of C++; some features are prohibited and a few extensions are outlined. This language does not meet Shore's needs for a language-independent type specification language that would support access and manipulation of a single database by application programs written in a variety of languages.

The Shore project continues to monitor the progress of the ODMG standardization effort and hopes to reconcile SDL with ODL at some time in the future.

The current document is intended to be a reference manual rather than a tutorial introduction to SDL. See the document Getting Started with Shore for a detailed example that illustrates most of the features of SDL. The manual page *intro(sdl)* and other manual pages cited there describe the C++ language binding for Shore in detail.

Features that are not supported in the current release of SDL are INDICATED THIS WAY.

2 Lexical Syntax

The lexical syntax of SDL is identical to that of C++ with the exception that the sets of *keywords*, *operators*, and *other separators* are different. In particular, both forms of C++ comment are supported, and comments and whitespace are handled exactly as in C++.

Operators and separators are drawn from the following set.

```
%    )    ,    :    <    >    ]    |
&    *    -    ::   <<   >>   ^    }
(    +    /    ;    =    [    {    ~
```

Keywords are any of the following words, plus the word `int` [1].

[1] This list of keywords is automatically generated from the BNF for the language. Strictly speaking, `int` is not a token, but is special-cased by the scanner, but as far as users are concerned, it is a keyword.

```
all          export       long         sequence
any          external     lref         set
as           false        module       short
attribute    float        octet        string
bag          import       ordered_by   struct
boolean      in           out          switch
case         index        override     true
char         indexable    private      typedef
class        inout        protected    union
const        interface    public       unsigned
default      inverse      ref          use
double       list         relationship void
enum
```

3 Grammatical Conventions

SDL grammar rules are presented at the start of many of the following sections. They are also collected in Appendix A. The notation is an extended form of BNF, with "regular expressions" allowed on the right-hand side.

Terminal symbols (tokens) are represented by upper-case words or strings enclosed in single quotes. Non-terminal symbols (syntax classes) are denoted by lower-case words. Grammar operators are as follows.

word* : *spec —*word* is defined to be an instance of *spec*

***spec* +** —one or more instances of *spec*

***spec* *** —zero or more instances of *spec*

spec1 spec2 —an instance of *spec1* followed by an instance of *spec2*

spec1* | *spec2 —an instance of *spec1* or an instance of *spec2*

[***spec***] —an optional instance of *spec*

(***spec*** **)** —an instance of *spec*

The postfix operators '+' and '*' have higher precedence than concatenation, which has higher precedence than '|'. Colon (':') has the lowest precedence of all. Parentheses are used to override these precedences as appropriate. Indentation is used to indicate continuation of a grammar rule over multiple lines. A standard BNF grammar (acceptable to YACC or Bison) is given in Appendix B.

4 Objects and Values

All persistent state in Shore is encapsulated in *objects*. An object has *attributes*, which define its persistent state, *operations*, which may change its state and/or return information about its

current state, and *relationships*, which determine how it is related to other objects. An object also has a unique *identity*; that is, modifying the state of an object is different from creating a new object with the new state. Objects do not nest. Although a particular application may define a "part-subpart" relationship between two objects, each object has an independent identity and lifetime. A *value* is a unit of stored state. In contrast with objects, values may be nested but do not have independent identities. The state of an object is a value, as is the state of each attribute of the object and each component of a structured value.

Each object has a fixed-size *core* and an optional variable-size *extension*. The structure of the core is defined by an *interface definition* in SDL. The extension supports variable-sized strings and sequences. The user can also use the extension as a storage pool to allocate values dynamically and link them to the core and to each other with *local references* (lrefs) which are converted to pointers (memory addresses) when they are brought into memory. Shore automatically brings objects into memory as an application traverses relationships. Large objects are faulted into memory incrementally, a page at a time. Each lref is adjusted as necessary to reflect the storage address used to hold the page of the object to which it refers.

5 Modules

```
specification : module*
module : 'module' ID '{' mod_export* mod_import*
        module_member* '}' [';']
mod_export : 'export' ( ID | 'all' ) ';'
mod_import : 'use' module_name [ 'as' ID ] ';'
    | 'import' module_name ';'
module_name : STRING_LITERAL
module_member : const_dcl ';'
    | type_dcl ';'
    | interface_dcl ';'
scoped_name : ID ( '::' ID )*
```

An SDL definition is a sequence of *module definitions*. Multiple module declarations can be combined in one source file, but they are separately processed as if they were in separate source files. In other words, there is no special scope associated with a file. The result of compiling a module definition is a *module object* in the database. A module defines a name scope (see the following section) in which names are bound to constants, types, and interfaces. These names are exported if they appear in the `export` declaration at the start of the module; otherwise, they are local and can only be used in the module in which they are defined. [2]

The declaration `export all` exports all "top level" names defined in the module. The declaration `use` *module_name* `as` *ID*; makes all exported names defined in the named module (as well as the types, constants, etc. that they designate) available in the current module. The *module_name* is a file name of a module object stored in the database, expressed as a

[2]Note that it is only the names (identifiers) that are local, not the things to which they refer. A constant, type, etc. whose name is not exported might nonetheless be referenced outside the module through another name or expression.

string literal. It is interpreted relative to a path given as a command-line argument to the SDL compiler (see the manual page *sdl(sdl)*).

Each exported name in that module can be used in the current module as *ID::name*, where *ID* is the identifier specified in the `use` declaration. If no *ID* is specified, the name of the module (as indicated in the module header) is used.

The *import* declaration imports all names exported by the indicated module as if they were defined in the current module. That is, they can be used without any *ID::* prefix if there is no ambiguity. (Importing a module M does not import the modules imported by M; it imports only the names exported by M, except if M exports "all.")

The module name given on an *import* declaration can be an identifier or a quoted path name. An identifier `mod` is treated the same as `"mod"`, and the module is expected to be found in the list of directories determined by command-line arguments to the compiler.

5.1 Examples

```
module constants {
    // installed as "constants" in the
    // directory determined by the first -d flag if present,
    // /types otherwise.
    export TitleSize;
    const long CharacterWidth = 1;
    const long TitleSize = 40*CharacterWidth;
}

module mod1 {
    export all;
    use "constants" as C;
    typedef char Title[C::TitleSize];
}

module mod2 {
    export all;
    import "constants";
    typedef char header[TitleSize]; // NB: TitleSize is 40!
}
```

The *import* declarations must follow the *export* declarations.

6 Linking

A module may refer to names defined in other modules with the aid of `use` and `import` declarations. If a module is compiled before one or more of the modules to which it refers, the resulting module object will be *incomplete*. An incomplete module is analogous to a object module (a ".o file") with undefined external references. The *SDL linker* takes a set of incomplete modules that

CONTAIN ONLY REFERENCES TO EACH OTHER AND TO COMPLETE MODULES, AND RESOLVES THE REFERENCES.

7 Name Scopes

The body of a `module`, `interface`, `struct`, or `union` declaration is a *name scope*. Each scope contains a sequence of *definitions*, each of which defines one or more names and binds each of them to a constant, type, interface, attribute, or operation. Names defined in a scope must be distinct. Each scope also has a name. Name *N* defined in scope *S* can be referred to unambiguously with the `scoped_name` *S::N*. Since scopes nest, this construction may be iterated. For example, *S::T::N* refers to the name *N* defined in scope *T*, which was defined in scope *S*. In this document, "name" means unqualified name (one with no "::") unless specified otherwise. A use of an unqualified name *N* refers to the definition of *N* in the smallest scope enclosing the use that defines *N* (but see also the discussion of inheritance in Section 8.1).

The `import` clause of a module and the `inheritance` clause of an interface effectively merge the sets of scoped names defined in two or more scopes. Names defined in more than one of these scopes must be qualified to prevent ambiguity.

7.1 Examples

```
module m1 {
    export all;
    const long C = 1;
}
module m2 {
    import "m1";
    const long D = C; // error: ambiguous (m1::C or m2::C)
    const long C = 2;
    const long E = m1::C; // ok: E = 1
}
```

8 Interfaces

```
interface_dcl : 'interface' ID [ interface_body ]
interface_body : [ inheritance ] '{' interface_dcls '}'
inheritance : ':' parent ( ',' parent )*
parent : access_spec scoped_name
access_spec : 'private' | 'protected' | 'public'
interface_dcls : ( access_spec ':' interface_members )*
interface_members : interface_member*
interface_member : const_dcl ';'
    | type_dcl ';'
    | attr_dcl ';'
    | relationship_dcl ';'
    | op_dcl ';'
```

```
    | override ';'
override : 'override' scoped_name ( ',' scoped_name )*
```

An *interface* definition is similar to a class definition in C++. The *access_specs* preceding names of parent interfaces and introducing groups of member declarations have exactly the same meaning as in C++, except that they are not optional. The form *"interface ID;* (with no `interface_body`) denotes a *"forward-declaration*; the complete declaration of the interface must appear elsewhere in the same module.

A *const_dcl* defines a constant value. A *type_dcl* defines a type, which may be simple (integer, float, etc.) or structured. Types are described in more detail below. An *attr_dcl* introduces an *attribute*, which corresponds to an "instance variable" in SmallTalk or a "data member" in C++. The attributes of an object define its state. A `relationship_dcl` defines a special kind of attribute called a *relationship*, which establishes a correspondence between objects.

An *op_dcl* introduces an *operation*. Operations are similar to "methods" in SmallTalk or "function members" in C++. SDL defines only the *signatures* of operations. They are bound to executable code in a language-dependent manner. An *override* does not introduce a new member, but indicates that the named members (which must designate inherited operations) will be bound to code in this class that overrides the binding in ancestor classes. (A similar feature is in Modula 3. C++ requires that the entire function header be repeated, but that it exactly match the header in base class from which the function member is inherited. The proposed C++ standard relaxes the requirement for an exact match, allowing the return type of the overriding function to be derived from the return type of the function it is overriding. Introduction of a similar feature into SDL is for further study.)

Override applies only to operations; it is an error to name attributes or any members other than operations in an *override*.

8.1 Example

```
interface AtomicPart {
public:
    attribute long               id;
    attribute char               type[TypeSize];
    attribute long               buildDate;
    attribute long               x, y;
    attribute long               docId;

    attribute set<Connection>    to;   // to connection objs
    attribute set<Connection>    from; // back pointers

    attribute ref<CompositePart> partOf; // up pointer

    void swapXY();
    void toggleDate();
    void DoNothing() const;

    long traverse(in BenchmarkOp op,
```

```
            inout PartIdSet visitedIds) const;
    void init(in long ptId, in ref<CompositePart> cp);
    void Delete();
};
```

An interface is a name space, as defined in the previous section. The complete set of names defined by an interface includes the names defined by the interface's `members` as well as names defined by its parents (if any). The complete set of definitions is formed by taking the set-theoretic union of sets of scoped names. Thus members with the same name introduced in different interfaces can be distinguished by their scoped names, and a given member inherited from the one ancestor by multiple paths contributes only one member to a derived interface. (In C++ terminology, this means that all inheritance is effectively "virtual.") Each name defined by a (member of an) interface *hides* definitions of that name inherited from its parents. A use of an unqualified name inside an interface refers to the definition of that name in the interface or one of its ancestors that hides all other definitions. If no definition hides all others, the name is ambiguous and must be qualified. (This corresponds to the so-called "dominance rule" of C++). If there is no definition of the name in the scope containing the use, the use identifies a definition in an enclosing scope (if any).

SDL does not support operator (method) overloading.

8.2 More Examples

```
interface A {
    public:
        const long a = 1;
        const long b = 2;
};

interface B : public A {
    public:
        const long c = b; //  c = B::b = 3
        const long b = 3;
};

interface C : public A {
    public:
        const long c = 4;
};

interface D : public A, public C {
    public:
        const long d = 5;
        // D defines A::a, A::b, B::b, B::c,
        //        C::c, and D::d
        // Inside D, a, b, and d can be used without
        // qualification, but c must be qualified.
```

```
        // The name b resolves to B::b, since its
        // definition hides A::b, but neither B::c
        // nor C::c hides the other.
};
```

9 Constants

```
const_dcl : 'const' const_type ID '=' const_exp
const_type : integer_type | boolean_type |
        floating_pt_type | 'string'
        | type_name // must designate a simple type
integer_type : [ 'unsigned' ] ( 'long' | 'short' )
boolean_type : 'boolean'
floating_pt_type : 'float' | 'double'
const_exp : exp1 ( '|' exp1 )*
exp1 : exp2 ( '^' exp2 )*
exp2 : exp3 ( '&' exp3 )*
exp3 : exp4 ( shift_op exp4 )*
exp4 : exp5 ( add_op exp5 )*
exp5 : exp6 ( mul_op exp6 )*
exp6 : [ unary_op ] atom
atom : const_name | literal | '(' const_exp ')'
shift_op : '<<' | '>>'
add_op : '+' | '-'
mul_op : '*' | '/' | '%'
unary_op : '+' | '-' | '~'
literal :
    INTEGER_CONSTANT
    | STRING_LITERAL
    | CHARACTER_CONSTANT
    | FLOATING_CONSTANT
    | 'true' | 'false'
const_name : scoped_name // must designate a const
```

A *const_dcl* binds a name to a simple (unstructured) constant value. [3] Constants may be defined by C-like expressions involving literals and names of other constants. The type of the constant is explicitly indicated. An integer type must be explicitly indicated as *long* or *short*; there is no default integer size. Only one precision of floating point is supported.

9.1 Examples

```
const long Kilobyte = 1<<10;
```

[3] Note that the name is bound to the *value* of the expression, not the expression itself. If the expression contains names defined in other modules, the value may not be resolved until all the modules are linked together. See the discussion of linking above.

```
const long Megabyte = 1<<20;
const long BytesPerPage = 4*Kilobyte;
const long MemSize = 20*Kilobyte;
const long MaxPages = MemSize / BytesPerPage;
const float PI = 3.1415926525;
const float Avogadro = 6.02E24;
const string Message = "Error";
```

10 Types

```
type_dcl : 'typedef' type_spec declarators
    | struct_type | union_type | enum_type | external_type
type_spec : simple_type | constructed_type
simple_type : atomic_type
    | string_type
    | enum_type
    | ref_type
    | type_name
constructed_type : struct_type
    | union_type
    | sequence_type
atomic_type : floating_pt_type
    | integer_type
    | char_type
    | boolean_type
    | octet_type
    | any_type
type_name : scoped_name
declarators : declarator ( ',' declarator )*
declarator : ID [ array_size ]
array_size : '[' positive_int_const ']'
positive_int_const : const_exp
        // must evaluate to a positive integer

octet_type : 'octet'
char_type : 'char'
any_type : 'any'

struct_type : 'struct' ID [ struct_body ]
struct_body : '{' struct_member* '}'
struct_member : type_spec declarators ';'

union_type : 'union' ID [ union_body ]
union_body : 'switch' '(' discriminator ')' '{' case* '}'
discriminator : scalar_type ID
```

```
scalar_type : integer_type
    | char_type
    | boolean_type
    | enum_type
    | type_name // denoting a scalar type
case : case_label+ ( type_spec declarators ';' )+
case_label : 'case' const_exp ':' | 'default' ':'

enum_type : 'enum' ID '{' ID ( ',' ID )* '}'
sequence_type :
    'sequence' '<' type_name [ ',' positive_int_const ] '>'
string_type :
    'string' [ '<' positive_int_const '>' ]

external_type : 'external' external_qualifier ID
external_qualifier : 'typedef' | 'class' | 'enum' | 'union' | 'struct'
```

Types are either *simple* or *constructed*. Simple types include the atomic types (integer, floating-point, etc.), character strings, and enumerations. The *positive_int_const* in a *string* type bounds the maximum length of the string. If the constant is omitted, the length of the string is unbounded.

Constructed types are built from base types, which may themselves be simple or constructed. A *struct* is a fixed heterogeneous sequence of values, selected by field names. It corresponds to a Pascal *record* or a C *struct*. A *union* consists of a discriminator value of a scalar type and a heterogeneous sequence of values (as in a *struct*) whose types are determined by the value of the discriminator. A *sequence* is a homogeneous sequence of values of a base type. As with *strings*, a bound on the length is optional.

Beware the construction `sequence<ref<T>>`*, as the two adjacent ">>" characters are parsed as a right-shift operator. In order to avoid this problem, you must put a space between the two brackets: "> >".*

Arrays are as in C: one-dimensional, fixed-length sequences of values of the base type, whose elements are indexed by non-negative integers. As in C (and C++), the fact that a name denotes an array type is indicated in its definition by following it with a size in brackets. *References* ARE EITHER *local* OR *remote*. LOCAL REFERENCES ARE SIMILAR TO POINTERS IN OTHER LANGUAGES, BUT ARE CONSTRAINED TO POINT TO VALUES IN THE OBJECT CONTAINING THE REFERENCE. A remote reference is a "pointer" to zero, one, or several objects. A reference that appears at the "top level" of an object, called a *relationship*, may be paired with an *inverse*.

Each *type_decl* defines one or more names and binds them to types (or, in the case of `enum`, values). As in C++, the exact semantics is rather convoluted because the `typedef` syntax is optional for *struct*, *union*, and *enum* types. [4] The names defined by a *type_dcl* include the identifiers in its *declarators*, as well as the first *ID* in each *struct* or *union*, and all the *ID*s in each *enum*.

[4] This description may need more work. It is intended to convey the *same* semantics as in the corresponding fragment of C++.

A `type_dcl` of the form

typedef*S D*;

where the *S* is a `struct_type`, `union_type`, or `enum_type` and *D* is a list of `declarators`, is equivalent to the pair of `type_dcl`s

S;

typedef*N D*;

where *N* is the first `ID` in *S*. A `type_dcl` beginning with the keyword `typedef` binds each `ID` appearing in the `declarators` to a type: An `ID` without an `array_size` is bound to the type *T* denoted by the `type_spec`. An `ID` followed by an `array_size` that evaluates to a value `N` (which must be a positive integer) is bound to the type *array [0..N-1] of T*.

A `type_dcl` that is a `struct_type`, `union_type`, or `enum_type` binds the first `ID` to the type denoted by the `type_spec`. An `enum_type` also defines each `ID` inside the braces and binds it to a (distinct) constant. Note that an `enum_type` is not a name scope; the constant names are defined in the enclosing scope. In particular, the same name may not appear in two *enum* declarations in the same scope.

10.1 Examples

```
typedef long vector[100];
const long MaxName = 40;
struct FullName {
    string<MaxName> given_name, family_name;
    char initial;
};
typedef struct FullName2 {
    string<MaxName> given_name, family_name;
    char initial;
} FullName3;
struct PersonalInfo {
    FullName name;
        struct Addr {
        string number, name;
        string city;
        char state[2];
        long zip;
    } address;
};
enum WidgetType { Simple, Complex };
struct simple_case {
    float cost;
    string description;
};
struct complex_case {
```

```
    short part_count;
    sequence<Widget> components;
};
union Widget switch (WidgetType part_type) {
    case Simple:
        simple_case si;
    case Complex:
        complex_case cx;
};
```

These declarations define the types *vector*, *FullName*, *FullName2*, *FullName3*, *PersonalInfo*, *WidgetType*, and *Widget*, and the constants *MaxName* (with type *long* and value 40), *Simple*, and *Complex* (enumeration values of type *WidgetType*). The names *FullName2* and *FullName3* are aliases for a type that is equivalent to (but distinct from) the type bound to *FullName*.

10.2 External Types

All data types that represent persistent data must be defined in the SDL sources, but data types for arguments to operations may be defined externally. The names of these data types must be declared in the SDL sources with the appropriate declaration, as in the example below:

10.3 Examples

```
module exeg {
    external class a;
    external enum b;
    external typedef c;
    external union d;
    external struct e;

    interface eg {
        // ...
    public:
        e op(in a _a, in b _b, out c _c, out d _d);
    };
}
```

These declarations appear in the same places that any other type declarations may appear, but externally defined types can be used only in operations declarations. [5] See Section 14 for more information about operations.

Depending on the language for which a binding is generated, types declared to be external in the SDL source might have to be defined before the language binding header file is included in the application program source.

[5] The language binding determines the usefulness of the external declarations. Only things that have sensible bindings to "forward declarations" in the target language have any language binding generated as a result of an external declaration. For example, in a C++ binding, only "struct", "union", and "class" are given forward declarations in the language binding.

11 Attributes

```
attr_dcl : [ 'indexable' ] 'attribute' type_spec declarators
```

An `attr_dcl` defines one or more *attributes* of the enclosing `interface`. The attribute names defined by the `attr_dcl` are the `ID`s appearing in the `declarators`. The type of each attribute is the type that would have been bound to the name if the keyword `attribute` were replaced by `typedef`. If the `type_spec` is a `constructed_type`, the `attr_dcl` is equivalent to a `type_dcl` followed by a `attr_dcl`; that is,

`attribute`*S D*;

where *S* is a `constructed_type`, is equivalent to

S;

`attribute` *N D*;

where *N* is the type defined by *S*. THE WORD `indexable` INDICATES THAT THIS ATTRIBUTE MAY BE USED IN THE `indexed_by` CLAUSE OF A RELATIONSHIP (SEE THE NEXT SECTION).

12 Indexes

```
type_spec : 'index' '<' simple_type_spec ',' simple_type_spec '>'
```

Attributes can be indexes; these are called "manual indexes," meaning that they are explicitly intialized, loaded, and manipulated by an application program.

```
typedef     long ssn;     // social security number
interface Person {
public:
    attribute   string name;
    attribute   ssn    social_security_number;
};

interface IndexObject {
public:
    attribute index<string,Person> name_to_person;
    attribute index<ssn,string>    ssn_to_name;
};
```

The *index* declaration `index<string,Person>` is equivalent to `index<string,ref<Person> >`; the two declaration styles can be used interchangeably.

13 References and Relationships

```
ref_type    : local_ref | remote_ref
local_ref   : 'lref' '<' type_name '>'
```

```
remote_ref  : ref_kind '<' type_name '>'
ref_kind    : 'ref' | 'set' | 'bag' | 'list'
relationship_dcl : 'relationship' ref_kind
        '<' type_name '>' [ 'inverse' scoped_name ]
        [ 'ordered_by' scoped_name ]
```

References are similar to pointers in a memory-based language such as Pascal or C. A *local_reference*(*lref* for short) is a pointer to another value in the same object. The *type_name* can designate any value type. A *remote_reference* is a set of zero or more pointers to other objects. The *type_name* must designate an object type (*interface*), called the *target type* of the relationship. A *ref* points to zero or one objects, a *set* points to zero or more distinct objects, and a *bag* or *list* points to zero or more objects that are not necessarily distinct (that is, a bag or list contains the identity of each object zero or more times). LISTS ARE NOT SUPPORTED.

A *relationship* is similar to an attribute of reference type, but it may also be paired with an inverse relationship. A relationships is more restricted than a pointer in C or C++ in that it may only "point to" the start of a whole object, and it can only appear as a "top level" attribute. If the `inverse` clause is present, its `scoped_name` must designate a relationship of the target type, and that relationship must specify this relationship as its inverse. If the `scoped_name` does not start with *T::*, where *T* is the target type (the `type_name` in the `relationship_dcl`), a prefix of *T::* is implied.

The implementation will guarantee the integrity of inverse relationships. That is, given

```
interface A {
  ...
  relationship R<B> x inverse y;
  ...
};
interface B {
  ...
  relationship R<A> y inverse x;
  ...
};
```

where `R` is a `rel_kind`, and given objects `oA` and `oB` of types `A` and `B`, respectively, the invariant is that `oA.x` contains a reference to `oB` if and only if `oB.y` contains a reference to `oA`.

THE `ordered_by` CLAUSE MAY BE USED ONLY WITH `list` RELATIONSHIPS. THE FOLLOWING `scoped_name` MUST DESIGNATE AN ATTRIBUTE OF THE TARGET CLASS WHOSE TYPE SUPPORTS A '<' COMPARISON. (AS WITH THE `inverse` CLAUSE, AN INITIAL *target_type::* IS OPTIONAL). IT INDICATES THAT THE REFERENCES IN THE LIST SHOULD BE MAINTAINED IN ORDER, SORTED BY THE VALUES OF THE INDICATED ATTRIBUTE OF THE OBJECTS TO WHICH THEY REFER. IF THIS CLAUSE IS NOT SPECIFIED FOR `list` RELATIONSHIP, THE ORDERING IS THE CHRONOLOGICAL ORDER IN WHICH REFERENCES ARE ADDED TO IT.

14 Operations

```
op_dcl : result_type ID '(' [ parameters ] ')' [ 'const' ]
```

```
result_type : type_spec | 'void'
parameters : parameter (',' parameter)*
parameter : mode type_spec declarator
mode : 'in' | 'out' | 'inout'
```

An *op_dcl* defines the input/output signature of an operation of an object. The actual code that implements the operation is specified in a language-dependent manner. An operation accepts zero or more values as arguments and optionally returns a value a result. The type of the result, if any, is indicated by the *result_type*; if the *result_type* is `void`, the operation does not return a result. A parameter mode of `in` indicates that the corresponding argument is not modified by the operation; otherwise, the argument must be a reference to a region of memory capable of storing a value of the indicated type, and the operation may modify the contents of that region. (The region may be, but does not have to be, part of a cached copy of an object). A parameter mode of `out` indicates that the effect of the operation is independent of the initial contents of the memory region supplied as an argument. The word `const` indicates that the operation does not change the state of the containing object.

15 Appendix A: Collected Grammar

This section includes the entire SDL grammar, expressed in the notation described above in the **Grammatical Conventions** section.

See Appendix B for a version of the grammar acceptable to YACC.

```
// ================= sdlman.tex : 193
specification : module*
module : 'module' ID '{' mod_export* mod_import*
        module_member* '}' [';']
mod_export : 'export' ( ID | 'all' ) ';'
mod_import : 'use' module_name [ 'as' ID ] ';'
    | 'import' module_name ';'
module_name : STRING_LITERAL
module_member : const_dcl ';'
    | type_dcl ';'
    | interface_dcl ';'
scoped_name : ID ( '::' ID )*
// ================= sdlman.tex : 413
interface_dcl : 'interface' ID [ interface_body ]
interface_body : [ inheritance ] '{' interface_dcls '}'
inheritance : ':' parent ( ',' parent )*
parent : access_spec scoped_name
access_spec : 'private' | 'protected' | 'public'
interface_dcls : ( access_spec ':' interface_members )*
interface_members : interface_member*
interface_member : const_dcl ';'
    | type_dcl ';'
    | attr_dcl ';'
    | relationship_dcl ';'
    | op_dcl ';'
    | override ';'
override : 'override' scoped_name ( ',' scoped_name )*
// ================= sdlman.tex : 603
const_dcl : 'const' const_type ID '=' const_exp
const_type : integer_type | boolean_type |
        floating_pt_type | 'string'
        | type_name // must designate a simple type
integer_type : [ 'unsigned' ] ( 'long' | 'short' )
boolean_type : 'boolean'
floating_pt_type : 'float' | 'double'
const_exp : exp1 ( '|' exp1 )*
exp1 : exp2 ( '^' exp2 )*
exp2 : exp3 ( '&' exp3 )*
exp3 : exp4 ( shift_op exp4 )*
exp4 : exp5 ( add_op exp5 )*
exp5 : exp6 ( mul_op exp6 )*
exp6 : [ unary_op ] atom
atom : const_name | literal | '(' const_exp ')'
shift_op : '<<' | '>>'
add_op : '+' | '-'
mul_op : '*' | '/' | '%'
```

```
unary_op : '+' | '-' | '~'
literal :
    INTEGER_CONSTANT
    | STRING_LITERAL
    | CHARACTER_CONSTANT
    | FLOATING_CONSTANT
    | 'true' | 'false'
const_name : scoped_name // must designate a const
// ================= sdlman.tex : 674
type_dcl : 'typedef' type_spec declarators
    | struct_type | union_type | enum_type | external_type
type_spec : simple_type | constructed_type
simple_type : atomic_type
    | string_type
    | enum_type
    | ref_type
    | type_name
constructed_type : struct_type
    | union_type
    | sequence_type
atomic_type : floating_pt_type
    | integer_type
    | char_type
    | boolean_type
    | octet_type
    | any_type
type_name : scoped_name
declarators : declarator ( ',' declarator )*
declarator : ID [ array_size ]
array_size : '[' positive_int_const ']'
positive_int_const : const_exp
        // must evaluate to a positive integer

octet_type : 'octet'
char_type : 'char'
any_type : 'any'

struct_type : 'struct' ID [ struct_body ]
struct_body : '{' struct_member* '}'
struct_member : type_spec declarators ';'

union_type : 'union' ID [ union_body ]
union_body : 'switch' '(' discriminator ')' '{' case* '}'
discriminator : scalar_type ID
scalar_type : integer_type
    | char_type
```

```
    | boolean_type
    | enum_type
    | type_name // denoting a scalar type
case : case_label+ ( type_spec declarators ';' )+
case_label : 'case' const_exp ':' | 'default' ':'

enum_type : 'enum' ID '{' ID ( ',' ID )* '}'
sequence_type :
    'sequence' '<' type_name [ ',' positive_int_const ] '>'
string_type :
    'string' [ '<' positive_int_const '>' ]

external_type : 'external' external_qualifier ID
external_qualifier : 'typedef' | 'class' | 'enum' |
        'union' | 'struct'

// ================= sdlman.tex : 1019
attr_dcl : [ 'indexable' ] 'attribute' type_spec declarators
// ================= sdlman.tex : 1079
type_spec : 'index' '<' simple_type_spec ','
        simple_type_spec '>'
        // ================= sdlman.tex : 1114
ref_type    : local_ref | remote_ref
local_ref   : 'lref' '<' type_name '>'
remote_ref  : ref_kind '<' type_name '>'
ref_kind    : 'ref' | 'set' | 'bag' | 'list'
relationship_dcl : 'relationship' ref_kind
        '<' type_name '>' [ 'inverse' scoped_name ]
        [ 'ordered_by' scoped_name ]
// ================= sdlman.tex : 1249
op_dcl : result_type ID '(' [ parameters ] ')' [ 'const' ]
result_type : type_spec | 'void'
parameters : parameter (',' parameter)*
parameter : mode type_spec declarator
mode : 'in' | 'out' | 'inout'
```

16 Appendix B: Grammar in YACC Syntax

This section contains a version of the SDL grammar that is somewhat less readable than the version in Appendix A, but which can be directly processed by YACC (or Bison). It is automatically generated from the version in Appendix A.

```
specification:  module_list
;
module: MODULE ID '{' mod_export_list mod_import_list
        module_member_list '}' SEMICOLON_opt
         ;
mod_export:  EXPORT ID_or_ALL ';'
;
mod_import:  USE module_name AS_ID_opt ';'
| IMPORT module_name ';'
;
module_name:  STRING_LITERAL
;
module_member:  const_dcl ';'
| type_dcl ';'
| interface_dcl ';'
;
scoped_name:  ID COLON_COLON_ID_list
;
interface_dcl:  INTERFACE ID interface_body_opt
;
interface_body:  inheritance_opt '{' interface_dcls '}'
;
inheritance:  ':' parent COMMA_parent_list
;
parent:  access_spec scoped_name
;
access_spec:  PRIVATE
| PROTECTED
| PUBLIC
;
interface_dcls:  access_spec_COLON_interface_members_list
;
interface_members:  interface_member_list
;
interface_member:  const_dcl ';'
| type_dcl ';'
| attr_dcl ';'
| relationship_dcl ';'
| op_dcl ';'
| override ';'
;
override:  OVERRIDE scoped_name COMMA_scoped_name_list
;
const_dcl:  CONST const_type ID '=' const_exp
;
const_type:  integer_type
```

```
| boolean_type
| floating_pt_type
| STRING
| type_name
;
integer_type:  UNSIGNED_opt LONG_or_SHORT
;
boolean_type:  BOOLEAN
;
floating_pt_type:  FLOAT
| DOUBLE
;
const_exp:  exp1 BAR_exp1_list
;
exp1:  exp2 CARET_exp2_list
;
exp2:  exp3 AND_exp3_list
;
exp3:  exp4 shift_op_exp4_list
;
exp4:  exp5 add_op_exp5_list
;
exp5:  exp6 mul_op_exp6_list
;
exp6:  unary_op_opt atom
;
atom:  const_name
| literal
| '(' const_exp ')'
;
shift_op:  LESS_LESS
| GREATER_GREATER
;
add_op:  '+'
| '-'
;
mul_op:  '*'
| '/'
| '%'
;
unary_op:  '+'
| '-'
| '~'
;
literal:  INTEGER_CONSTANT
| STRING_LITERAL
```

```
| CHARACTER_CONSTANT
| FLOATING_CONSTANT
| TRUE
| FALSE
;
const_name:  scoped_name
;
type_dcl:  TYPEDEF type_spec declarators
| struct_type
| union_type
| enum_type
| external_type
;
type_spec:  simple_type
| constructed_type
;
simple_type:  atomic_type
| string_type
| enum_type
| ref_type
| type_name
;
constructed_type:  struct_type
| union_type
| sequence_type
;
atomic_type:  floating_pt_type
| integer_type
| char_type
| boolean_type
| octet_type
| any_type
;
type_name:  scoped_name
;
declarators:  declarator COMMA_declarator_list
;
declarator:  ID array_size_opt
;
array_size:  '[' positive_int_const ']'
;
positive_int_const:  const_exp
;
octet_type:  OCTET
;
char_type:  CHAR
```

```
;
any_type:  ANY
;
struct_type:  STRUCT ID struct_body_opt
;
struct_body:  '{' struct_member_list '}'
;
struct_member:  type_spec declarators ';'
;
union_type:  UNION ID union_body_opt
;
union_body:  SWITCH '(' discriminator ')' '{' case_list '}'
;
discriminator:  scalar_type ID
;
scalar_type:  integer_type
| char_type
| boolean_type
| enum_type
| type_name
;
case: case_label_list1
        type_spec_declarators_SEMICOLON_list1
         ;
case_label:  CASE const_exp ':'
| DEFAULT ':'
;
enum_type:  ENUM ID '{' ID COMMA_ID_list '}'
;
sequence_type: SEQUENCE '<' type_name
        COMMA_positive_int_const_opt '>'
         ;
string_type:  STRING LESS_positive_int_const_GREATER_opt
;
external_type:  EXTERNAL external_qualifier ID
;
external_qualifier:  TYPEDEF
| CLASS
| ENUM
| UNION
| STRUCT
;
attr_dcl:  INDEXABLE_opt ATTRIBUTE type_spec declarators
;
type_spec: INDEX '<' simple_type_spec ',' simple_type_spec
        '>'
```

```
        ;
ref_type:  local_ref
| remote_ref
;
local_ref:  LREF '<' type_name '>'
;
remote_ref:  ref_kind '<' type_name '>'
;
ref_kind:  REF
| SET
| BAG
| LIST
;
relationship_dcl: RELATIONSHIP ref_kind '<' type_name '>'
        INVERSE_scoped_name_opt
        ORDERED_BY_scoped_name_opt
         ;
op_dcl:  result_type ID '(' parameters_opt ')' CONST_opt
;
result_type:  type_spec
| VOID
;
parameters:  parameter COMMA_parameter_list
;
parameter:  mode type_spec declarator
;
mode:  IN
| OUT
| INOUT
;
module_list:
module_list module
| /* empty */
;
mod_export_list:
mod_export_list mod_export
| /* empty */
;
mod_import_list:
mod_import_list mod_import
| /* empty */
;
module_member_list:
module_member_list module_member
| /* empty */
;
```

```
SEMICOLON_opt:
';'
| /* empty */
;
ID_or_ALL:
ID
| ALL
;
AS_ID_opt:
AS ID
| /* empty */
;
COLON_COLON_ID_list:
COLON_COLON_ID_list COLON_COLON ID
| /* empty */
;
interface_body_opt:
interface_body
| /* empty */
;
inheritance_opt:
inheritance
| /* empty */
;
COMMA_parent_list:
COMMA_parent_list ',' parent
| /* empty */
;
access_spec_COLON_interface_members_list:
access_spec_COLON_interface_members_list access_spec ':'
        interface_members
         | /* empty */
;
interface_member_list:
interface_member_list interface_member
| /* empty */
;
COMMA_scoped_name_list:
COMMA_scoped_name_list ',' scoped_name
| /* empty */
;
UNSIGNED_opt:
UNSIGNED
| /* empty */
;
LONG_or_SHORT:
```

```
LONG
| SHORT
;
BAR_exp1_list:
BAR_exp1_list '|' exp1
| /* empty */
;
CARET_exp2_list:
CARET_exp2_list '^' exp2
| /* empty */
;
AND_exp3_list:
AND_exp3_list '&' exp3
| /* empty */
;
shift_op_exp4_list:
shift_op_exp4_list shift_op exp4
| /* empty */
;
add_op_exp5_list:
add_op_exp5_list add_op exp5
| /* empty */
;
mul_op_exp6_list:
mul_op_exp6_list mul_op exp6
| /* empty */
;
unary_op_opt:
unary_op
| /* empty */
;
COMMA_declarator_list:
COMMA_declarator_list ',' declarator
| /* empty */
;
array_size_opt:
array_size
| /* empty */
;
struct_body_opt:
struct_body
| /* empty */
;
struct_member_list:
struct_member_list struct_member
| /* empty */
```

```
;
union_body_opt:
union_body
| /* empty */
;
case_list:
case_list case
| /* empty */
;
case_label_list1:
case_label_list1 case_label
| case_label
;
type_spec_declarators_SEMICOLON_list1:
type_spec_declarators_SEMICOLON_list1 type_spec
        declarators ';'
         | type_spec declarators ';'
;
COMMA_ID_list:
COMMA_ID_list ',' ID
| /* empty */
;
COMMA_positive_int_const_opt:
',' positive_int_const
| /* empty */
;
LESS_positive_int_const_GREATER_opt:
'<' positive_int_const '>'
| /* empty */
;
INDEXABLE_opt:
INDEXABLE
| /* empty */
;
INVERSE_scoped_name_opt:
INVERSE scoped_name
| /* empty */
;
ORDERED_BY_scoped_name_opt:
ORDERED_BY scoped_name
| /* empty */
;
parameters_opt:
parameters
| /* empty */
;
```

```
CONST_opt:
CONST
| /* empty */
;
COMMA_parameter_list:
COMMA_parameter_list ',' parameter
| /* empty */
;
```