omniidl — The omniORB IDL Compiler

Duncan Grisby
AT&T Laboratories Cambridge

June 2000

1 Introduction

This manual describes omniidl, the omniORB IDL compiler. It is intended for developers who wish to write their own IDL compiler back-ends, or to modify existing ones. It also documents the design of the compiler front-end for those poor souls who have to track the IDL specification.

If you just wish to use omniidl to create stubs for C++ or Python, you should read the omniORB or omniORBpy manuals instead of this one.

1.1 Requirements

Back-ends for omniidl are written in Python, so to use it you must have an up-to-date Python interpreter. You must also understand Python to be able to follow this manual and write back-ends. You can download Python and associated documentation from `http://www.python.org/`.

The front-end scanner and parser are written using flex and bison; the rest of the front-end is written in C++. The code intentionally avoids using any advanced (and useful) features of C++, such as templates, so as to make it as portable as possible.

1.2 Running omniidl

On all platforms, there is a command named omniidl. On Unix platforms, omniidl is a Python script which runs Python via the `#!` mechanism. On Windows NT, there is an executable named `omniidl.exe`.

The omniidl command line has the form:

`omniidl` [*options*] `-b`<*back-end*> [*back-end options*] <*file 1*> <*file 2*> ...

The supported flags are:

`-D`*name*[=*value*]	Define *name* for the preprocessor.
`-U`*name*	Undefine *name* for the preprocessor.
`-I`*dir*	Include *dir* in the preprocessor search path.
`-E`	Only run the preprocessor, sending its output to stdout.
`-Y`*cmd*	Use *cmd* as the preprocessor, rather than the normal C preprocessor.
`-N`	Do not run the preprocessor.
`-T`	Use a temporary file, not a pipe, for preprocessor output.
`-Wp`*arg*[,*arg*...]	Send arguments to the preprocessor.
`-b`*back-end*	Run the specified back-end. For the C++ ORB, use `-bcxx`.
`-Wb`*arg*[,*arg*...]	Send arguments to the back-end.
`-nf`	Do not warn about unresolved forward declarations.
`-k`	Keep comments after declarations, to be used by some back-ends.
`-K`	Keep comments before declarations, to be used by some back-ends.
`-C`*dir*	Change directory to *dir* before writing output files.
`-i`	Run the front end and back-ends, then enter the interactive loop.
`-d`	Dump the parsed IDL then exit, without running a back-end.
`-p`*dir*	Use *dir* as a path to find omniidl back-ends.
`-V`	Print version information then exit.
`-u`	Print usage information.
`-v`	Verbose: trace compilation stages.

If you do not specify any back-ends (with the `-b` flag), omniidl just runs the compiler front-end, checking that the IDL is valid. If you specify more than one back-end, the back-ends are run in turn on the abstract syntax tree of each file. This permits you to generate stubs for more than one language in a single run. It also permits you to write back-ends which annotate or modify the abstract syntax tree to be used by later back-ends.

For example, the command:

```
omniidl -bdump -bpython foo.idl bar.idl
```

first reads and parses `foo.idl`, and runs the `dump` and `python` back-ends on it in turn. Then it reads and parses `bar.idl` and runs the two back-ends on that.

1.3 Preprocessor interactions

IDL is processed by the C preprocessor before omniidl parses it. Unlike the old IDL compiler, which used different C preprocessors on different platforms, omniidl always uses the GNU C preprocessor (which it builds with the name omnicpp). The `-D`, `-U`, and `-I` options are just sent to the preprocessor. Note that the current directory is not on the include search path by default—use '`-I.`' for that. The `-Y` option can be used to specify a different preprocessor to omnicpp. Beware that line directives inserted by other preprocessors are likely to confuse omniidl.

1.3.1 Windows 9x

The output from the C preprocessor is normally fed to the omniidl parser through a pipe. On some Windows 98 machines (but not all!) the pipe does not work, and the preprocessor output is echoed to the screen. When this happens, the omniidl parser sees an empty file, and produces useless stub files with strange long names. To avoid the problem, use the '`-T`' option to create a temporary file between the two stages.

1.4 Forward-declared interfaces

If you have an IDL file like:

```
interface I;
interface J {
  attribute I the_I;
};
```

then omniidl will normally issue a warning:

```
test.idl:1: Warning: Forward declared interface `::I' was never
fully defined
```

It is illegal to declare such IDL in isolation, but it *is* valid to define interface `I` in a separate file. If you have a lot of IDL with this sort of construct, you will drown under the warning messages. Use the `-nf` option to suppress them.

1.5 Comments

By default, omniidl discards comments in the input IDL. However, with the `-k` and `-K` options, it preserves the comments for use by the back-ends.

The two different options relate to how comments are attached to declarations within the IDL. Given IDL like:

```
interface I {
  void op1();
  // A comment
  void op2();
};
```

the `-k` flag will attach the comment to `op1()`; the `-K` flag will attach it to `op2()`.

1.6 Interactive loop

When omniidl is given the `-i` option, it runs the compiler front-end and any back-ends specified, and then drops into Python's interactive command loop. Within the interactive loop, you can `import omniidl`. The parsed AST is then available as `omniidl.idlast.tree`. This mode is useful for investigating the parsed tree.

1.7 Copyright

All parts of omniidl are licensed under the GNU General Public License, available in the file `COPYING`.

As a special exception to the terms of the GPL, we do not consider back-ends to be derived works of omniidl. This means that you may distribute back-ends you write under any terms you like. The back-ends we distribute are licensed under the GPL, so you must abide by its terms if you distribute or modify our back-ends.

As another exception, we do not consider the output of the back-ends we distribute to be derived works of those back-ends. You may therefore use generated stubs with no restrictions.

2 Back-end interface

There are three elements to the back-end interface: requirements on the back-end modules themselves, a set of output and utility functions, and the interface to the parsed IDL.

2.1 Back-end modules

omniidl back-ends are just normal Python modules. When you specify a back-end with `-bfoo`, omniidl first tries to open the Python module named `omniidl_be.foo`. If that fails, it tries to open the module just named `foo`, using the normal `PYTHONPATH` mechanism. As with any Python module, the module `foo` can either be implemented as a single file named `foo.py`, or as a directory `foo` containing a file named `__init__.py`.

The only requirement on back-end modules is that they contain a function with the signature `run(tree, args)`, where `tree` is an `AST` object as described in section 2.3.3, and `args` is a list of argument strings passed to the back-end.

Back-ends may also optionally provide a variable named `cpp_args` which contains a list of strings containing arguments to be given to the C preprocessor. For example, the Python back-end contains the line:

```
cpp_args = ["-D__OMNIIDL_PYTHON__"]
```

2.2 Output and utility functions

The purpose of most back-ends is to output source code in some language. It is often the case that much of the output is independent of the specifics of the IDL input. The output for an IDL interface, for example, might be an extensive class definition containing configuration and initialisation code which is largely independent of the specifics of the interface. At various places throughout the class definition, there would be items which *were* dependent on the interface definition.

omniidl supports this with *template* based output functions. Templates are simply strings containing the code to be output, including expressions surrounded by '@' characters. When the templates are output, the keys inside the '@' expressions are replaced with values according to the output arguments. An '@' symbol can be output by putting '@@' in the template.

The output facilities are provided in the `omniidl.output` module by the `Stream` class. The primary method of `Stream` objects is `out()`, which takes arguments of a template string and a set of key/value pairs to be used in `@` substitutions. For example, if `st` is a `Stream` object, then the code:

```
  template = """\
class @id@ {
public:
  @id@(@type@ a) : a_(a) {}
private:
  @type@ a_;
};"""

  st.out(template, id="foo", type="int")
```

would result in output:

```
class foo {
public:
  foo(int a) : a_(a) {}
private:
  int a_;
};
```

When `@` expressions are substituted, the expression is actually *evaluated*, not just textually replaced. This means that you can write templates containing strings like '`@obj.name()@`'. Expressions must evaluate to strings. This feature should not be over-used—it is very easy to write incomprehensible template expressions. The vast majority of templates should only use simple string substitutions.

Commonly, it is necessary to nest definitions which are output inside other definitions. `Stream` objects keep track of a current indentation level to aid this. The methods `inc_indent()` and `dec_indent()` increment and decrement the current indent level respectively. The number of spaces corresponding to a single indent level is configured when the `Stream` is created. Occasionally, you may need to output code which ignores the current indent level (preprocessor directives in C, for example). The `niout()` method is identical to `out()` except that it performs no indentation.

The `Stream` constructor takes two arguments, a file opened for writing, and an integer specifying how many spaces to use for each indent level.

omniidl.output.Stream

Stream(file,indent_size)
Initialise a Stream with the given output file and indent size.

inc_indent()
Increment the indent level.

dec_indent()
Decrement the indent level.

out(template,key=val,...)
Output the template string template with key/value substitution and indenting.

niout(template,key=val,...)
As out(), but with no indenting.

2.2.1 Utility functions

The omniidl.idlutil module contains a number of useful functions:

omniidl.idlutil

escapifyString(str)
Convert any non-printable characters in string `str` into octal escape sequences.

pruneScope(target,from)
Given two scoped names represented as lists of strings, return `target` with any prefix it shares with `from` removed. For example:

```
>>> pruneScope(['A','B','C','D'],['A','B','D'])
['C','D']
```

relativeScope(from,dest)
Given two globally-scoped name lists, return a minimal scoped name list which identifies the destination scope, without clashing with another identifier. If the only valid result is a globally-scoped name, the result list is prefixed with `None`.

slashName(sn,from)

dotName(sn,from)

ccolonName(sn,from)
Prune scoped name list `sn` with `pruneScope(sn,from)`, then convert into a string with name components separated by '/', '.' or '::'.

2.3 Abstract Syntax Tree

The main meat of the back-end interface is in the `omniidl.idlast` and `omniidl.idltype` modules. When the compiler parses an IDL file, it creates a tree of objects representing the IDL declarations. The classes for these declarations are defined in the `idlast` module. The way an IDL declaration is split into objects closely follows the terms within the IDL grammar presented in chapter 3 of the CORBA 2.3 specification.

2.3.1 Visitor pattern

All objects within the back-end interface support the *visitor* pattern. They have an `accept(visitor)` method which acts on a visitor adhering to the interfaces in the `omniidl.idlvisitor` module. Note that Python's dynamic type system means that visitor objects need not actually derive from the classes defined in `idlvisitor`[1]. Also note that you do not have to use the visitor pattern if you do not wish to.

[1]It is even possible to use a Python module as a visitor object.

2.3.2 Pragmas and comments

Any unknown `#pragmas` encountered in the IDL are attached to nodes within the AST. Similarly, comments are attached if omniidl is run with the `-k` or `-K` fields.

`omniidl.idlast.Pragma`

`text()`
: Text of the pragma.

`__str__()`
: Same as `text()`.

`file()`
: File containing the pragma.

`line()`
: Line within the file.

`omniidl.idlast.Comment`

`text()`
: Text of the comment.

`__str__()`
: Same as `text()`.

`file()`
: File containing the comment.

`line()`
: Line within the file.

2.3.3 The root of the tree

The back-end's `run()` function (described in section 2.1) is passed an object of class `AST`.

omniidl.idlast.AST

file()
The file name of the main IDL file being compiled.

declarations()
List of `Decl` objects corresponding to declarations at file scope.

pragmas()
List of `Pragma` objects containing `#pragmas` which occurred before any declarations. Later `#pragmas` are attached to `Decl` objects.

comments()
List of `Comment` objects containing comments which were not attached to declarations (see section 1.5).

accept(visitor)
Visitor pattern accept.

2.3.4 Base declaration

All declarations in the tree are derived from the `Decl` class:

omniidl.idlast.Decl

file()
The name of the file in which this declaration was made.

line()
The line number within the file.

mainFile()
Boolean: true if the declaration is in the main IDL file; false if it is in an included file.

pragmas()
List of `Pragma` objects containing `#pragmas` which occurred after this declaration, but before any others.

comments()
List of `Comment` objects containing comments attached to this declaration (see section 1.5).

accept(visitor)
Visitor pattern accept.

2.3.5 Declarations with a repository identifier

Some classes of declaration object also inherit from the `DeclRepoId` mixin class:

---

omniidl.idlast.DeclRepoId

identifier()
Name of the declaration as a string.

scopedName()
List of strings forming the fully-scoped name of the declaration. e.g. `::foo::bar::baz` is represented as `['foo','bar','baz']`.

repoId()
Repository identifier of the declaration.

2.3.6 Declaration classes

The declaration objects making up the tree have the following classes:

---

omniidl.idlast.Module (Decl,DeclRepoId)
Module declaration

definitions()
List of `Decl` objects declared within this module, in the order they were declared.

continuations()
List containing `Module` objects which are continuations of this module. When modules are re-opened, multiple `Module` objects with the same name appear in the enclosing `Module` or `AST` object. In case it's useful, the first `Module` object for a particular module has a list containing continuations of that module. You will probably not have any use for this.

---

omniidl.idlast.Interface (Decl,DeclRepoId)
Interface declaration

abstract()
Boolean: true if the interface is declared abstract.

inherits()
List of interfaces from which this one inherits. Each list member is either an `Interface` object, or a `Declarator` object belonging to a typedef to an interface.

contents()
List of `Decl` objects for all items declared within this interface.

declarations()
Subset of `contents()` containing types, constants and exceptions.

callables()
Subset of `contents()` containing `Operations` and `Attributes`.

---

omniidl.idlast.Forward (Decl,DeclRepoId)
Forward-declared interface

abstract()
Boolean: true if the interface is declared abstract.

fullDecl()
`Interface` object corresponding to the full interface declaration or `None` if there is no full declaration.

---

omniidl.idlast.Const (Decl,DeclRepoId)

Constant declaration

constType()

`idltype.Type` object of the constant. Aliases not stripped.

constKind()

TypeCode kind of the constant with aliases stripped. So for a constant declared with:

```
typedef long MyLong;
const MyLong foo = 123;
```

`constKind()` will return `tk_long`, but `constType()` will return an `idltype.Declared` object (see page 20) which refers to `MyLong`'s typedef `Declarator` object.

value()

Value of the constant. Either an integer or an `Enumerator` object.

---

omniidl.idlast.Declarator (Decl,DeclRepoId)

Declarator used in typedefs, struct members, attributes, etc.

sizes()

List of array sizes, or `None` if it is a simple declarator.

alias()

`Typedef` object that the declarator is part of, or `None` if the object is not a typedef declarator.

---

omniidl.idlast.Typedef (Decl)

Typedef declaration

aliasType()

`idltype.Type` object that this is an alias to.

constrType()

Boolean: true if the alias type was constructed within this typedef declaration, like

```
typedef struct foo { long l; } bar;
```

declarators()

List of `Declarator` objects.

omniidl.idlast.Member (Decl)
Member of a struct or exception

memberType()
`idltype.Type` object for the type of this member.

constrType()
Boolean: true if the member type was constructed within the member declaration. e.g.

```
struct S {
  struct T {
    long l;
  } the_T;
};
```

declarators()
List of `Declarator` objects.

omniidl.idlast.Struct (Decl,DeclRepoId)
Struct declaration

members()
List of `Member` objects for the struct contents.

recursive()
Boolean: true if the struct is recursive, e.g.

```
struct S {
  long l;
  sequence <S> ss;
};
```

omniidl.idlast.Exception (Decl,DeclRepoId)
Exception declaration

members()
List of `Member` objects for the exception contents.

omniidl.idlast.CaseLabel (Decl)
One label within a union

default()
Boolean: true if this is the default label.

value()
Label value. Either an integer or an `Enumerator` object. For the default case, returns a value used by none of the other union labels.

labelKind()
TypeCode kind of the label.

omniidl.idlast.UnionCase (Decl)
One case within a union

labels()
List of `CaseLabel` objects.

caseType()
`idltype.Type` object for the case type.

constrType()
Boolean: true if the case type was constructed within the case.

declarator()
`Declarator` object

omniidl.idlast.Union (Decl,DeclRepoId)
Union declaration

switchType()
`idltype.Type` object corresponding to the switch type.

constrType()
Boolean: true if the switch type was declared within the switch statement. Only possible for Enums.

cases()
List of `UnionCase` objects.

recursive()
Boolean: true if the union is recursive.

omniidl.idlast.Enumerator (Decl,DeclRepoId)
Enumerator of an enum

No non-inherited functions.

omniidl.idlast.Enum (Decl,DeclRepoId)
Enum declaration

enumerators()
List of `Enumerator` objects.

omniidl.idlast.Attribute (Decl)
Attribute declaration

readonly()
Boolean: true if the attribute is read only.

attrType()
`idltype.Type` object for the attribute's type.

declarators()
List of `Declarator` objects for this attribute. All declarators are guaranteed to be simple.

identifiers()
Convenience function returning a list of strings containing the attribute identifiers from the declarators. e.g. for the declaration

```
attribute long a, b;
```

`identifiers()` will return `['a','b']`.

`omniidl.idlast.Parameter (Decl)`
Parameter of an operation

`direction()`
Integer: 0 == in, 1 == out, 2 == inout.

`is_in()`
Boolean: true if in or inout.

`is_out()`
Boolean: true if out or inout.

`paramType()`
`idltype.Type` object for the parameter type.

`identifier()`
String containing the parameter identifier.

`omniidl.idlast.Operation (Decl,DeclRepoId)`
Operation declaration

`oneway()`
Boolean: true if the operation is one way.

`returnType()`
`idltype.Type` object for the return type.

`parameters()`
List of `Parameter` objects.

`raises()`
List of `Exception` objects which the operation can raise.

`contexts()`
List of strings declared as context for the operation.

`omniidl.idlast.Native (Decl)`
Native declaration

Native should not be used in normal IDL.

No non-inherited functions.

---

omniidl.idlast.StateMember (Decl)
State member of a concrete valuetype

memberAccess()
Integer: 0 == public, 1 == private.

memberType()
`idltype.Type` object for member type.

constrType()
Boolean: true if the member type is declared within the StateMember.

declarators()
List of `Declarator` objects.

---

omniidl.idlast.Factory (Decl)
Factory method of a valuetype

identifier()
String containing the factory identifier.

parameters()
List of `Parameter` objects.

---

omniidl.idlast.ValueForward (Decl,DeclRepoId)
Forward-declared valuetype

abstract()
Boolean: true if declared abstract.

fullDecl()
`Value` or `ValueAbs` object corresponding to the full valuetype declaration or `None` if there is no full declaration.

---

omniidl.idlast.ValueBox (Decl,DeclRepoId)
Boxed valuetype declaration

boxedType()
`idltype.Type` object for the boxed type.

constrType()
Boolean: true if boxed type is declared inside the valuetype declaration.

---

omniidl.idlast.ValueAbs (Decl,DeclRepoId)
Abstract valuetype declaration

inherits()
List of `ValueAbs` objects from which this inherits.

supports()
List of `Interface` objects which this valuetype supports.

contents()
List of `Decl` objects for all items defined within this valuetype.

declarations()
Subset of `contents()` containing types, constants and exceptions.

callables()
Subset of contents() containing `Operations` and `Attributes`.

omniidl.idlast.Value (Decl,DeclRepoId)
Valuetype declaration

custom()
Boolean: true if declared custom.

inherits()
List of valuetypes from which this inherits. The first may be a `Value` object or a `ValueAbs` object; any others will be `ValueAbs` objects.

truncatable()
Boolean: true if the inherited `Value` is declared truncatable; false if not, or there is no inherited `Value`.

supports()
List of `Interface` objects which this valuetype supports.

contents()
List of `Decl` objects for all items defined within this valuetype.

declarations()
Subset of `contents()` containing types, constants and exceptions.

callables()
Subset of `contents()` containing `Operations`, `Attributes`, `StateMembers` and `Factorys`.

2.3.7 Type objects

All type objects are derived from the base class `Type`:

---

omniidl.idltype.Type
Base class for types

kind()
TypeCode kind of type.

unalias()
Return an equivalent `Type` object with top-level aliases stripped. Only has an effect with typedef types.

accept(visitor)
Visitor pattern accept.

The basic CORBA types (null, void, short, long, unsigned short, unsigned long, float, double, boolean, char, octet, any, TypeCode, Principal, long long, unsigned long long, long double, and wide char) are represented by objects of type `omniidl.idltype.Base`, derived from `Type`, with no extra methods.

The template types—string, wstring, sequence, and fixed—do not have associated `Decl` objects since they are not explicitly declared. They are always implicitly declared as part of another declaration.

---

omniidl.idltype.String (Type)
String type

bound()
Bound of a bounded string, or 0 for unbounded strings.

---

omniidl.idltype.WString (Type)
Wide string type

bound()
Bound of a bounded wstring, or 0 for unbounded wstrings.

---

omniidl.idltype.Sequence (Type)
Sequence type

seqType()
`idltype.Type` object representing what the sequence contains.

bound()
Bound of a bounded sequence, or 0 for unbounded sequences.

---

omniidl.idltype.Fixed (Type)

Fixed point type

digits()
Number of digits in number.

scale()
Scale of number.

All other types (interface, struct, union, enum, typedef, exception, valuetype) must be explicitly declared. They are represented with `Declared` objects:

---

omniidl.idltype.Declared (Type)

Explicitly declared type

decl()
`omniidl.idlast.Decl` object which corresponds to this type.

scopedName()
Fully scoped name of the type as a list of strings.

name()
Simple name of the type, i.e. the last element of the scoped name.

2.3.8 Finding a named Decl

Normally, back-ends walk over the tree of `Decl` objects, dealing with the declarations as they encounter them. Occasionally, however, it may be useful to find a declaration by its scoped name. Only `Decl`s which inherit from `DeclRepoId` can be found in this way.

---

omniidl.idlast

findDecl(scopedName)
Find the `Decl` object which has the scoped name list `scopedName`. If a declaration with the specified name does not exist, the `DeclNotFound` exception is raised.

2.4 An example back-end

The following code is an extremely simple back-end which just prints the names of all operations declared within an IDL file. Unfortunately, it is so simple that it does not show many features of the back-end interface. You should look at the `dump.py` and `python.py` back-ends for a more extensive example.

```
from omniidl import idlast, idlvisitor, idlutil
```

```
import string

class ExampleVisitor (idlvisitor.AstVisitor):

    def visitAST(self, node):
        for n in node.declarations():
            n.accept(self)

    def visitModule(self, node):
        for n in node.definitions():
            n.accept(self)

    def visitInterface(self, node):
        name = idlutil.ccolonName(node.scopedName())

        if node.mainFile():
            for c in node.callables():
                if isinstance(c, idlast.Operation):
                    print name + "::" + \
                          c.identifier() + "()"

def run(tree, args):
    visitor = ExampleVisitor()
    tree.accept(visitor)
```

The visitor object simple recurses through the `AST` and `Module` objects, and prints the operation names it finds in `Interface` objects.

Note that since `AstVisitor` (and similarly `TypeVisitor` which is not used in the example) has all operations declared to be no-ops, the `ExampleVisitor` class does not have to declare visit functions for all node types. This can be a disadvantage if your back-end is supposed to perform some action for all node types, since there will be no error if you accidentally miss a node type. In those situations it is better to declare a visitor class which does not derive from the visitor base classes.

3 Front-end architecture

Not here yet.