A CORBA IDL compiler for SWI-Prolog

Jan Wielemaker
SWI,
University of Amsterdam
The Netherlands
E-mail: jan@swi.psy.uva.nl

January 25, 2001

Abstract

This document contains a brief overview of a CORBA IDL compiler for (SWI-)Prolog. The compiler automatically generates the interface code between Prolog and an external ORB. It has been used with a wide variety of ORB implementations using either the plain C or the C++ language binding of the ORB.

The current development uses Orbacus 4.0.4.

Contents

1 Introduction

CORBA is the emerging standard for distributed computing. It makes components of a distributed system independent on the platform and language as well as accesible over the network.

A component in a distributed system is described using an IDL, (*Interface Description Language*) module. IDL is an object oriented language. It describes the component in terms of *interfaces* (classes in normal OO systems), which have attributes and methods. Method arguments are typed. The type system of IDL contains the usual primitives: various length integers, floating point numbers and strings, as well as constructs for aggregate types: (un)bounded sequences, arrays, structures and unions (*switch-type*).

A *language binding* describes how constructs from the IDL are mapped onto constructs of the target language. Well defined bindings exist for various languages, including C, C++, Java and Lisp. No such binding is defined for Prolog.

This document describes an evolving binding and its implementation for Prolog.

2 Architecture

There is a wide range of implementations of the C and C++ bindings available. As this project was carried out with limited resources, it appeared natural to base the implementation on these bindings. Using an existing binding, we avoid involvement in the broker and protocol layers of CORBA. The architecture is outlined in figure 1.

3 The Prolog mapping

A mapping describes what the IDL looks like from the target language. In this section we will go through all mapped IDL construcs.

3.1 Types

3.1.1 Scalar Types

All *scalar integer* types ((un)signed long, (un)signed short, char, octed are mapped into Prolog integers. If an argument is passed to the CORBA interface the Prolog integer should satisfy the limitations of the IDL type. A floating point number representing an integer in the required range of the type is accepted too. If an argument is passed to Prolog, it always appears as a Prolog integer.

The types `float` and `double` are mapped into Prolog floating point numbers.

3.1.2 Booleans

The CORBA type CORBA_Boolean is mapped onto the Prolog constants `true` and `false`.

3.1.3 Enum Types

Enum types are mapped onto atoms holding the name of the member of the enum type.

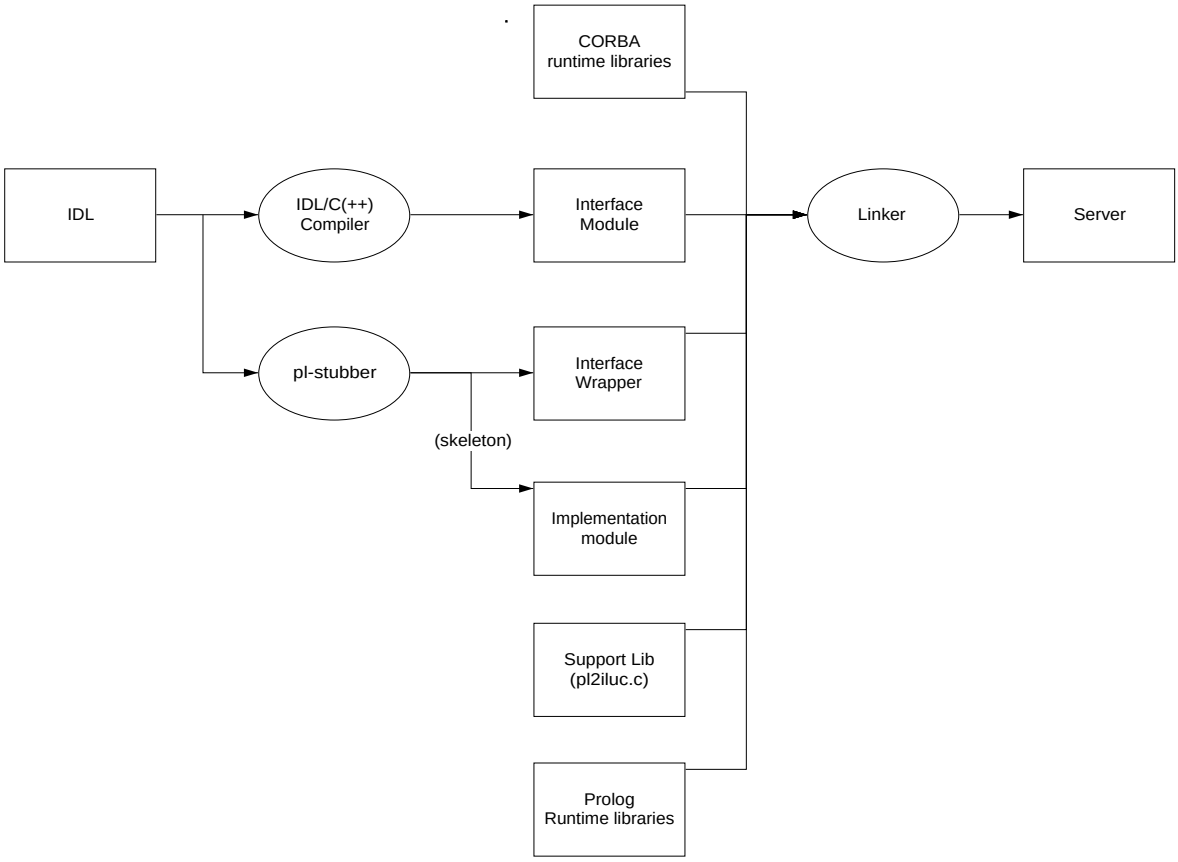

Figure 1: Architecture of the Prolog/CORBA binding

3.1.4 Strings

Strings are mapped onto atoms. This normally requires a Prolog implementation offering atom garbage collection.

3.1.5 Structures

A structure is mapped into a Prolog term. The principal functor is the name of the structure. The arguments of the term denote the fields of the structure by position.[1]

3.1.6 Sequences

Both bound and unbound sequences are mapped into Prolog lists.

3.1.7 Arrays

The current IDL compiler does not support arrays. On Prolog systems with unbound term-arity they could be mapped onto terms, providing `arg/3` for accessing the elements in a natural and efficient fashion. On other systems, trees may be appropriate. To ensure portability, the support libraries should include predicates to build and analyse arrays.

3.1.8 Switch types

An IDL `switch-type` is the combination of a union and a type identifier (scalar or enum). It is mapped onto a term named after the switch-type holding the type identifier and the actual value.

If the type identifier is an enum, a good alternative mapping would be $\langle Enum \rangle(\langle Value \rangle)$. This however does not allow for integer type identifiers.

3.2 Constants

Constants are not yet available to the implementation module. They are handled by the IDL compiler for type specifications. The IDL compiler should generate a predicate to access the constants. For example:

```
?- idl_constant(Name, Value).
```

3.3 Modules

IDL modules are mapped onto Prolog modules. The current compiler cannot deal with interfaces appearing outside modules or nested modules. In the future, the code for interfaces without a module should be in the Prolog user module and nested modules should be mapped onto Prolog modules where the name is the concatenation of the module-path, separated by underscores.

For a generic mapping, Modules cannot be used as modules are not yet part of the (ISO) Prolog standard.

[1] Terms are a natural and efficient method for representing structures, but they loose the naming of the fields as they appear in the IDL. The IDL compiler should generate one or more utility predicates for accessing the arguments in the term by name.

3.4 Interfaces

Interfaces are not mapped explicitely.

3.5 Methods

A method is mapped onto a predicate in the implementation module. The name of the predicate is constructed from the interface name and the method name, separated by an underscore.

The first argument to the predicate is a reference to the interface object. Next are the arguments describing the input/output arguments. At the server side, input arguments are bound and output arguments are unbound. At the client input arguments must be bound. For output arguments, the converted value is unified with the argument. Arguments of type `inout` are not yet supported. They will probably be passed as a term `inout`(*In*, *Out*). Finally, if the method has a return-type, this is appended as the last argument. The argument conventions for the return-value are the same as for output arguments.

3.6 Exceptions

IDL exceptions are mapped onto ISO Prolog catch/throw exception handling. The exception-term is a term whose name is the exception name. The arguments are mapped as structure arguments. A server raising an exception calls `throw/1` using the exception term. A client calls `catch/3` for handling these exceptions.

4 Other server implementation aspects

4.1 Creating interface instances

Instances are create using the predicate

⟨*interface*⟩_create_true(*+Marker*, *-Handle*)

If *Marker* is unbound, the created object is anonymous. If it is bound to an atom this name is used as a marker. The implication thereof depends on the ORB used.

- *Orbacus 4*
  The *Marker* is used to register this object with the given name at the *BootManager*. The object is now available under the URL

  `corbaloc:iiop:`⟨*host*⟩`:`⟨*port*⟩`/`⟨*name*⟩

  This URL can be passed to `corba_string_to_object/2` on the client-side. See `corba_initialize_server/2` on how to bind the server to a specified port.

4.1.1 Using the Orbix Loader

The Orbix loader is activated by the Orbix deamon to create instances of a specified interface with a specified marker. The loader calls the predicate user:`corba_loader/3` to create objects:

user:corba_loader(*+ModuleAndInterface, +Marker, -Object* **)**

Interface is of the form ⟨*Module*⟩_⟨*Interface*⟩. ⟨*Object*⟩ should be unified with a reference to an instance of the requested interface, normally by calling the appropriate `*_create_true` predicate.

It would be better to define a `corba_loader/3` predicate in each module and pass the plain interface name.

4.2 Overall control predicates

The Prolog mapping defines a number of predicates to deal with the overall control of the session, as well as other environment control. As the possibilities differ widely between the various CORBA implementations, this set of predicates is tentative.

corba_initialize_server(*+Options, -Server* **)**

Initialize the CORBA server library. *Options* is a list of options. *Server* is unified with a handle to the ORB. ORB implementations vary widely in the options that may be handed to initialising the server. The option list is described below:

Common options	
server(*ServerId*)	Denotes the identifier under which the server is registered.
ILU options	
protocol(*ProtocolId*)	The protocol used for communication. Default is the ILU internal protocol, `iiop_1_0_1` should be used to switch to the CORBA IIOP protocol.
transport(*Stack*)	Determines the transport layer(s) used. Default is Sun RPC, `['tcp_0_0']` should be used with IIOP to communicate to other brokers.
OmniBroker options	
port(*Port*)	Specify the TCP/IP port used for the server. Objects created with a *Marker* can be located using `corba_string_to_object/2` knowing the host, port and marker.
Orbix options	
timeout(*Seconds*)	After this idle-time, `corba_main_loop/1` will return. Can be used to exit from the server. The Orbix deamon will relaunch the server if a new request is made.

corba_main_loop(*+Server* **)**

Start processing requests to the server. This predicate does not return before the server is stopped.

4.3 Other utility predicates

corba_object_to_string(*+Object, -Atom* **)**

Returns a 'stringified object-reference' to *Object* as an atom in *Atom*, Stringified object references can be passed to another broker. The receiving broker can obtain a handle to the remote object using `corba_string_to_object/2`.

Stringified object references can be used to establish communication between any two 'IIOP' compliant broker implementations.

corba_string_to_object(*+String, -Object***)**
Obtain a handle to a remote interface object described by the given string. Fully portable is the 'IOR' as generated by `corba_object_to_string/2`.

For Orbacus *String* can also take the form of an URL. See section 4.1 for details.

corba_duplicate(*+Object, -Duplicate***)**
Create a duplicate of an object. See `CORBA::_duplicate()` in the CORBA documentation for details.

corba_release(*+Object***)**
Decrements the reference count of the object, releasing it if the reference count drops to zero.

5 Client specific aspects

This section discusses client-specific issues.

corba_initialise_orb(*+Options, -ORB***)**
Initialize the client-side the ORB for the client-side. *Options* is the same as for `corba_initialize_server/2`, though some options are only relevant for the server side. *ORB* is unified with a term containing handles to the initialized ORB and BOA objects.

5.1 Using the Orbix *binder*

The Orbix Class::_bind() call is mapped onto the Prolog predicate ⟨*interface*⟩`__bind/4`:

interface__bind(*-Object, +Marker, +Service, +Host***)**
Calls Orbix ⟨*interface*⟩::_bind(Marker, Service, Host). An marker that is an empty atom or a variable is taken as 'Any object belonging to this interface'. After using the interface, it should be released using `corba_release/1`.

6 Implementation

The Prolog IDL compiler is a Prolog program. Figure 2 outlines the design of the compiler.

The shell-script `pl-stubber` provides a simple toplevel for the program. Use 'pl-stubber –help' to see the commandline options.

7 Status

Initially, the IDL compiler was used in combination with the ILU CORBA/C mapping, for which we realised both the client and server side. Later we moved to the Omnibroker/C++ mapping for which we realised only the server mapping. Finally, we moved to Orbix/C++.

While changing platforms due to external requirements, we generalised the code generator to deal with subtle differences in the mappings. The tested ORB's are in table 1.

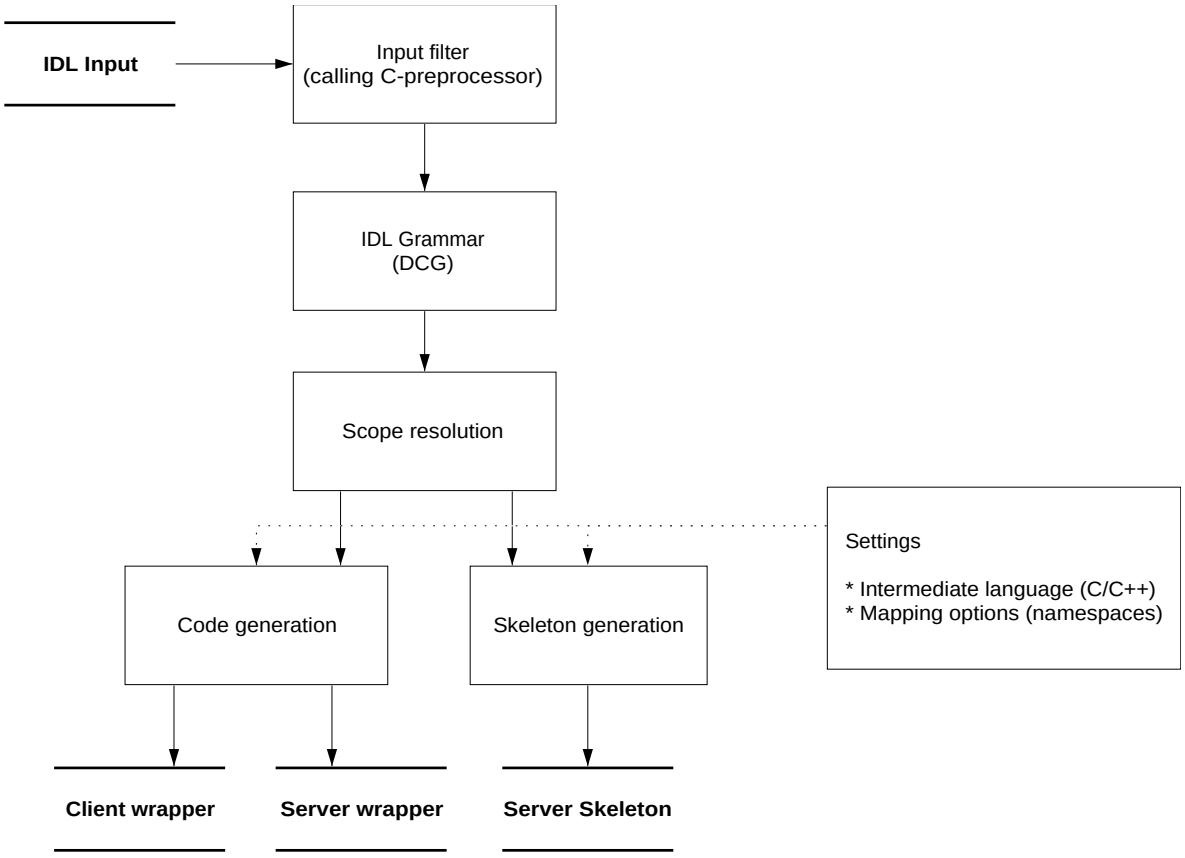

Figure 2: Modular design of the IDL compiler

ORB	**server**	**client**
ILU 2.0a10 (Linux and Solaris, gcc 2.7)	ok	ok
OmniBroker 1.0	ok	ok
Orbacus 4.04	ok	ok
Orbix 2.1	not tested	not implemented
Orbix 2.2 (NT 4.0, MSVC4.2)	ok	ok

Table 1: Supported ORB's

We intend to support a wide variety of ORB implementations, including both commercial and free versions.

7.1 The mapping

The outlined mapping was inspired by the CORBA/C mapping. Some aspects of the mapping need further investigation.

- *switch-type*
  As Prolog is a dynamically typed language, switch-types can often be avoided. For example a union of integer, float and string can appear either as a Prolog integer, float or an atom.

  The only problem occurs of two elements of the switch map onto the same Prolog datatype, and this information conveys semantic information.

- *structure*
  Predicates should be provided to extract and create structure terms using the field-names.

- *interface*
  A more natural appearance of the CORBA interface objects from Prolog can be achieved. Instead of writing

  ⟨*Module*⟩:⟨*Interface*⟩(Self, Args ..., Return)

  we could opt for

  corba_send(Self, Args ...)
  corba_get(Self, Args ..., Return)