SWI-Prolog C++ Proxy

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: wielemak@science.uva.nl

August 13, 2008

Abstract

This document describes an infrastructure for calling Prolog from C++ that allows for controlled access from C++ based on a generated *proxy* class. In our first target the proxy class talks to a multi-threaded Prolog server using TCP/IP sockets. In future versions this will be extended with SSL sockets, pipes and native DLL embedding. The C++ and Prolog sourcecode for all these scenarios is identical, except for the code that initialises the system.

Contents

1 Introduction

SWI-Prolog is well suitable for writing a wide range of complete applications without introducing other languages into the system and an even wider range when linking C/C++ coded shared objects to access external resources, in real-life Prolog is often embedded in systems written in other languages. The reasons vary. Re-use of existing code, expertise in the development team or external requirements are commonly encountered motivations.

Embedding Prolog is not a logical choice in our view. An important part of the power of Prolog can be found in its development system where retrying goals and reloading patched code on the running system speedup development considerably. In embedded system these features are often lost or severely limited due to lack of access to the interactive Prolog toplevel or inability of the rest of the application to stay synchronised with the dynamic nature of the Prolog part of the application.

If you have to embed there are several options for doing so, each with specific advantages and disadvantages.

- *Linking as library*
  Linking Prolog as a library is attractive as it allows for two-way communication at very low overhead. It is also the most complicated approach, often loosing access to the Prolog toplevel entirely, introducing possibly threading and (on POSIX systems) signal synchronisation problems, link conflicts and difficulty to localise bugs.

- *Using pipes*
  By using anonymous pipes between the hosting system and Prolog we introduce a seperation that makes it easier to localise problems and reliably stop and start Prolog. The price is -again- loosing the Prolog toplevel, slower communication and pipes only provide a single communication channal.

- *Using sockets*
  Using sockets connecting to a continuously running multi-threaded Prolog server does keep access to the Prolog toplevel, offers very short startup times and allows to distribute the applications over multiple hosts on the network. The price is that it is way harder to setup the communication (something must ensure the server is running and allocate a port for it) and the server must be written thread-safe.

2 Overview

This packages consists of the following components:

- *Interface definition*
  The library `cpp_interface.pl` and `typedef.pl` define directives that allow you to specify the predicates that are callable from C++ and their types. Only specified predicates can be called and only with matching types. Restricting what can be called greatly improves security when used in a server setting. Section 4 describes these declarations.

- *Code generation*
  The library `cpp_codegen.pl` defines the code generator. The code generator is used to create the C++ source for a proxy class that is used in the C++ client to talk to Prolog. Section 6 describes generating the C++ proxy.

- *Prolog server*
  When using sockets, the library `cpp_server.pl` defines the Prolog server. See section 8 for details.

- *C++ client library*
  The file `SWI-proxy.cpp` and `SWI-Proxy.h` provide the base classes for the client proxy.

3 Related techniques

The technique used in this package are not new nor unique. Inter-language communication has been a topic in ICT for a long period, resulting in various widespread and well established solutions. The power of this solution is that it is tailured to Prolog's features such as non-deterministic predicates, lightweight, simple and fast. The weakness is of course being bound to Prolog and, currently, C++. Proxy generators can be defined for other languages, reusing most of the infrastructure except for the details of the code generation.

- *CORBA*
  CORBA generates language specific proxies from a language neutral (IDL) specification. There are no bindings for Prolog. We once wrote a proxy generator between the C++ proxy and Prolog. This design is fairly elegant and produces fast interprocess communication. CORBA however is a complicated big system that require considerable resources for doing even the smallest tasks.

- *HTTP (optionally with SOAP)*
  Using the Prolog HTTP server is another alternative. HTTP provides the basic message *envelope*. The message content is still undefined. SOAP (an XML based content format) can be used here. Backtracking over solutions is hard to implement based on the stateless HTTP protocol. The approach is much more complicated and the various protocol layers require much more data and processing time. Experience show latency times of approx. a few milliseconds, where our server shows latency times of approx. 0.1 millisecond (AMD 1600+, SuSE Linux).

- *InterProlog*
  InterProlog is a stream-based connection to Java. I have no experience with it.

- *SWI-Prolog C++ interface*
  Using the native SWI-Prolog C++ interface does not provide network transparency and is much harder to program. The advantage is that it allows for mutual calling, more threading alternatives and many more.

3.1 Prolog Portability

The design can work with other Prolog systems. The server exploits multi-threading, but with some limitations this can be changed to run in a single thread. The proxy generator is portable with some effort and it is also possible to generate the proxy with SWI-Prolog and use it with a server written in another Prolog system. The proxy itself is pure C++, knowing nothing about Prolog.

4 Defining the interface

The interface definition defines the C++ callable predicates as well as their types and modes. The interface only deals with ground terms. The type language syntax is defined in the library `typedef.pl` and is based on the Mycroft/O'Keefe type language.

:- type(*TypeSpec***)**

If *TypeSpec* is of the form *Alias* = *Type*, *Alias* is an alias for the type named *Type*. If *TypeSpec* is of the form *Type* -> *Def*, *Def* is the definition of *Type*. Polymorphism is expressed using multiple definitions seperated by the `|` (vertical bar) symbol.[1] A single definition is a term whose arguments define the types of the arguments.

There are three *primitive* types: `integer`, `float` and `atom`.

Valid type declarations for our C++ interface do not use polymorphism and a fully expanded type definition consists of structures and primitive types. The argument *names* for compound types are derived from the type-name and usually bound to a real type using a type-alias. Here is an example:

```
:- type first_name = atom.
:- type last_name  = atom.
:- type age        = integer.

:- type person -> person(first_name, last_name, age).
```

The callable predicates are specified using the library `cpp_interface.pl`, which defines two directives.

:- cpp_callable(*Head [= Attributes], ...***)**

Defines *Head* to be callable from C++. *Head* has the same number of argument as the predicate that must be callable. Each argument is of the form *+Type* or *-Type* for resp. an *input* and *output* argument. *Attributes* is a list of attributes. Currently defined attributes are:

one

Predicate succeeds exactly ones. Failure is an error. If the predicate succeeds non-deterministically the choicepoints are discarded (cut). Such predicates are mapped to a `void` method on the C++ proxy class. If the predicate fails this is mapped to a C++ exception. This is the default.

zero_or_one

Predicates fails or succeeds ones. If the predicate succeeds non-deterministically the choicepoints are discarded (cut). Such predicates are mapped to an `int` method on the C++ proxy class returning `FALSE` if the predicate fails and `TRUE` if it succeeds.

zero_or_more

Predicate is non-deterministic. Such predicates are mapped to a subclass of class *PlQuery*.

as(*Name***)**

If present, the predicate is mapped to a C++ method or query class named *Name* instead of the name of the predicate. This allows for mapping different calling patterns of the same predicate to different C++ methods or classes.

[1] The design allows for limited polymorphism, but this is not yet part of the current implementation.

:- cpp_type(*CName = Functor*)

Specifies that the Prolog type *Functor* is represented by the C++ class CName. This allows for different naming conventions in the Prolog and C++ world.

The examples below depend on the type examples above.

```
:- cpp_callable
        version(-atom) = [one],
        find_person_younger_than(+age, -person) = [zero_or_more].

version('0.0').

find_person_younger_than(MaxAge, person(FirstName, LastName, Age)) :-
        person(FirstName, LastName, Age),
        Age =< MaxAge.
```

5 Compound data as seen from C++

Compound data that is to be communicated to Prolog is represented as a C++ class. This class must provide methods to fetch the components for use as a predicate input argument and with a method to create fill an instance of this class for predicate output arguments. These methods are:

`void` **initialize**(*t1 a1, t2 a2, ...*)

The initialize method is called with as many objects of the proper type as there are arguments in the Prolog term. The primitive types are `long`, (for Prolog integers) `double` (for Prolog floats) and the C++ *std* class `string` for atoms.

`Type` **get_*field***()

For each named field (see section 4) a function must be provided that extracts the field and returns the appropriate type. For atom typed fields the return value can be an std `string` or a plain C `char*`.

Below is a possible implementation for the above defined person class.

```
class person
{
public:
  char *first_name;
  char *last_name;
  int age;

  person()
  { first_name = NULL;
    last_name = NULL;
    age = -1;
  };
  ~person()
```

```
  { if ( first_name ) free(first_name);
    if ( last_name ) free(last_name);
  }

  char *get_first_name() const { return first_name; }
  char *get_last_name() const  { return last_name; }
  long  get_age() const  { return age; }

  void initialize(string fn, string ln, long years)
  { if ( first_name ) free(first_name);
    if ( last_name  ) free(last_name);

    first_name = strdup(fn.c_str());
    last_name  = strdup(ln.c_str());
    age = years;
  }
};
```

6 Generating the C++ proxy

The C++ proxy class is automatically generated from the Prolog declarations using the library `cpp_codegen.pl`. To generate the code load this library in a Prolog process that has all the `cpp_callable/1` and type declarations in place and invoke the predicate `cpp_server_code/2`:

cpp_server_code(+*File, +Options*)
: Generate the C++ proxy class to access the deterministic predicates and the query classes for the non-deterministic predicates and write them to the given *File*. *Options* consists of

  server_class(*Name*)
  : Name of the proxy class. If omitted it is called `MyProxy`.

7 Using the proxy classes

7.1 Passing primitive datatypes

Primitive data are the Prolog types integer, float and atom.

7.2 Passing compound data

Compound data is represented as a compound term in Prolog and, unless renamed using `cpp_type/2`, an equally named class in C++.

7.3 Non-deterministic queries

The proxy for a non-deterministic predicates is a subclass of `PlQuery`. The name of the class is the name of the predicate, unless modified using the `as`(*Name*) attribute with `cpp_callable/1`. A

query is started by creating an instance of this class using a pointer to the proxy as argument. The only method defined on this class is ::next_solution(). This method uses the same arguments as the proxy methods that represent deterministic queries. The following example fetches all functors with arity 3 defined in Prolog:

```
:- use_module(library(typedef)).
:- use_module(library(cpp_interface)).

:- cpp_callable
        current_functor(-atom, +integer) = [zero_or_more].
```

```
#include <iostream>
#include "myproxy.h>

int
main(int argc, char **argv)
{ MyProxy proxy("localhost", 4224);

  try
  { between q(&proxy);
    string name;

    while ( q.next_solution(name, 3) )
    { cout << name << endl;
    }
  } catch ( PlException &ex )
  { cerr << (char *)ex;
  }

  return 0;
}
```

7.4 Nesting queries

Non-deterministic queries are initiated by creating an instance of its class. The query is said to be *open* as long as the query object is not destroyed. New queries, both deterministic and non-deterministic can be started while another query is still open. The *nested* query however must be closed before more solutions can be asked from the query it is nested in.

The example below computes a table of all square roots for the numbers 1 to 100 using prolog to generate the numbers on backtracking using `between/3` and access to `sqrt/2`. First the Prolog code, followed by the C++ code.

```
:- use_module(library(typedef)).
:- use_module(library(cpp_interface)).

:- cpp_callable
```

```
        between(+integer, +integer, -integer) = [zero_or_more],
        sqrt(+float, -float).

sqrt(In, Out) :- Out is sqrt(In).
```

```
#include <iostream>
#include "myproxy.h>

int
main(int argc, char **argv)
{ SqrtProxy proxy("localhost", 4224);

  try
  { between q(&proxy);
    long l = 1;
    long h = 100;
    long i;

    while ( q.next_solution(l, h, i) )
    { double ifloat = (double)i;
      double rval;

      proxy.sqrt(ifloat, rval);
      cout << "sqrt(" << i << ") = " << rval << endl;
    }
  } catch ( PlException &ex )
  { cerr << ex;
  }

  return 0;
}
```

8 Running the server

For running the server we need a Prolog process with the actual predicates and their declarations loaded. We load the library `cpp_server` and invoke `cpp_server/1`:

cpp_server(+*Options***)**
: Start the C++ server in the current process. This creates a small thread with the alias `cpp_accept` that accepts new connections and, for each new connection, starts a new thread that handles the queries for the client. Options include:

  port(_Port_**)**
  : Port on which to bind the server. Default is 4224.

9 Putting it all together: a complete example

The base-classes for the runtime system are installed in the SWI-Prolog include directory as `SWI-proxy.cpp` and its header `SWI-proxy.h`. These files are *not* compiled into a library. Considering compatibility between different compilers and compilation models (threading, etc.) it is thought to be easier to include this code into the target project using the source-code.

The directory examples (installed as `.../pl/doc/packages/examples/cppproxy`) contains some stand-alone examples as well as a `README` explaining how to compile and run the examples.

10 Status

The current implementation is a demonstrator. Issues to be resolved in future versions of this package include

- *Handle arrays*
  Provide automatic conversion of C++ arrays and/or std library vectors to Prolog lists. Currently sets can be extracted from Prolog by enumerating a non-deterministic predicate and send to Prolog using repetitive calls. Both imply sending many small packages over the wire.
- *Authentication and security*
  Currently the server is 'wide open', Limiting the IP for connecting hosts is a first step. Other steps are login using password challenge/response. Sequence numbers to avoid man-in-the-middle attacks and the use of SSL.
- *Alternative communication channals*
  Currently only the TCP/IP version is implemented. See introduction.
- *Error recovery*
  Protocol errors (which can be caused by incompatible proxy and Prolog server type declarations) crash the connection. Re-synchronisation is difficult to implement. We could do a version check by computing a hash from the Prolog interface specification and validate this on communication startup.

10.1 Portability

The system is designed to be portable using any modern C++ compiler. It has been tested on Linux using g++ 3.3.4 and MS-Windows using MSVC 6.

11 Installation

11.1 Unix systems

Installation on Unix system uses the commonly found *configure*, *make* and *make install* sequence. SWI-Prolog should be installed before building this package. If SWI-Prolog is not installed as `pl`, the environment variable `PL` must be set to the name of the SWI-Prolog executable. Installation is now accomplished using:

```
% ./configure
% make
% make install
```

This installs the foreign library `serialize` in `$PLBASE/lib/$PLARCH` and the Prolog library files in `$PLBASE/library` and the files `SWI-proxy.cpp` and `SWI-proxy.h` in `$PLBASE/include`, where `$PLBASE` refers to the SWI-Prolog 'home-directory'.

11.2 Windows system

If you have successfully installed the system from source you can install this package using

```
% nmake /f Makefile.mak
% nmake /f Makefile.mak install
```

If not, compile serialize.c using the command below and install the files by hand or using the makefile after setting the variable PLBASE to the base of the installed Prolog system.

```
% plld -o serialize serialize.c
```

Index

between/3, 8

cpp_callable/1, 5, 7
cpp_server/1, 9
cpp_server_code/2, 7
cpp_type/1, 6
cpp_type/2, 7

get_*field*(), 6

initialize(), 6

PlQuery *class*, 5

sqrt/2, 8

type/1, 5