BinProlog 2006 11.x Professional Edition
BinProlog Interface Guide
Connecting BinProlog with C/C++, Java, Tcl/Tk

Paul Tarau

BinNet Corp.
WWW: http://www.binnetcorp.com
E-mail: binnetcorp@binnetcorp.com

July 7, 2008

1   BinProlog's C-interface

To be able to extend BinProlog and possibly embed it as logic engine in your no run-time fee application you will need a BinProlog C-source code license. See the files SOURCE.LICENSE and PRICING for more information. You can also interact with other languages under UNIX using the bidirectional pipe based interface starting from BinProlog's Tcl/Tk interfaced (see directory TCL).

The following sequence of quick examples shows how it works.

1.1   A Demo Program

BinProlog's C-interface together with a prototype example been moved in directory `c_inter`. Start by cloning this directory if you want to build an interface application and edit/adapt the files.

The discussion which follows basically explains the content of the files `c.c c.h and c.pl` in this directory.

Once BinProlog is correctly installed as file "bp" somewhere in your path along with the libraries bp.o and wam.o which should be in directory `../src` and you have a C-compiler around, you can simply do `make` in directory `c_inter` to create a standalone application which integrates the functionality of the C-code in file c.c.

To test the resulting standalone executable `c_binpro`, try out the demo `c_test` defined in file `c.pl`. Start running it by typing `c_binpro` and then type `c_test.` at the Prolog prompt. To avoid the prompt and directly launch the application add

```
main:-c_test.
```

in file `c.pl` before typing `make`.

1.2   Calling C from BinProlog: adding new builtins

New builtins are added in `headers.pl` and after a "make realclean; make" a new version of BinProlog containing them is generated automatically. Starting from `headers.pl` the system will first generate the files `defs.h, prof.h, builtins.pl` and then recompile itself. For this recompilation either a previous version of BinProlog[1]

or SICStus Prolog 2.x is used.

An example of declaration in `headers.pl` is:

```
 b0(+/3,arith(1),in_body).  <=== arity=3+1 by binarization
```

Notice that arith(1) means that it is "like an arithmetic functions" which returns *one* value, i.e this should be used as in

```
 +(0,1,Result).
```

I would suggest to start with the simplest form of interaction: a call of your own C-code from BinProlog. Try modifying in the file c.c (provided with the C-sources of BinProlog, in directory `c_inter`), the function `new_builtin()` which looks as follows:

---

[1]Use "make remake or make again" if BinProlog is the only Prolog around.

```
/* ADD YOUR NEW BUILTINS HERE

   they can be called from Prolog as:
   new_builtin(O,<INPUT_ARG>,<OUTPUT_ARG>)

   X(1) contains the integer 'opcode' of your builtin
   X(2) contains your input arg
   regs[I] contains somthing that will be unified with what you return
      (the precise value of I depends on the register allocator).

   You are expected to 'return' either
    - a non-null object that will be unified with <OUTPUT_ARG>, or
    - NULL to signal FAILURE

   As the returned object will be in a register this
   can be used for instance to add a garbage collector
   that moves every data area around...

*/

/**
  simple function to be called from BinProlog
*/
int sfun(char *s) {
   char buf[100];
   sprintf(buf,"=> CALLED sfun(char *) at %ld!",(unsigned long)sfun);
   strcat(s,buf);
   return 1;
}

/**
  simple float function to be called from BinProlog
*/
double ffun(double d) {
   char buf[100];
   sprintf(buf,"=> CALLED double<-ffun(double d) at %ld!",(unsigned long)ffun);
   return d*2.5;
}

term new_builtin(register term H,register term regs,register term *A,register instr P,register stack wam)
   BP_check_call();
   switch(BP_op)
  {
    /* for beginners ... */

    case 0:
       /* this just returns your input argument (default behavior) */
    break;

    case 1:
      BP_result=BP_integer(13); /* this example returns 13 */
    break;

    case 2:
```

```
    BP_result=BP_atom("hello"); /* this example returns 'hello' */
break;

/* for experts ... */

case 3: /* iterative list construction */
{ cell middle,last,F1,F2; int i;
  BP_make_float(F1, 77.0/2);
  BP_make_float(F2, 3.14);

  BP_begin_put_list(middle);
      BP_put_list(BP_integer(33));
      BP_put_list(F1);
      BP_put_list(BP_string("hello"));
      BP_put_list(F2);
  BP_end_put_list();

  BP_begin_put_list(last);
    for(i=0; i<5; i++) {
      BP_put_list(BP_integer(i));
    }
  BP_end_put_list();

  BP_begin_put_list(BP_result);
    BP_put_list(BP_string("first"));
    BP_put_list(middle);
    BP_put_list(last);
    BP_put_list(F1);
    BP_put_list(F2);
  BP_end_put_list();
} break;

case 4: /* cons style list construction */
  BP_begin_cons();
  BP_result=
   BP_cons(
      BP_integer(1),
      BP_cons(
          BP_integer(2),
          BP_nil
      )
   );
  BP_end_cons();
break;

case 5: /* for hackers only ... */ ;
  BP_result=(cell)H;

  H[0]=g.DOT;
  H[1]=X(2);

  H[2]=g.DOT;
  H[3]=BP_integer(99);
```

```
   H[4]=g.DOT;
   H[5]=(cell)(H+5); /* new var */

   H[6]=g.DOT;
   H[7]=(cell)(H+5); /* same var as previously created */

   H[8]=g.DOT;
   H[9]=BP_atom("that's it");

   H[10]=g.NIL;
   H+=11;
 break;

 case 6:
   BP_fail();
 break;

 case 7:
   { cell T=BP_input;
      if(BP_is_integer(T))
         {int i;
          BP_get_integer(T,i);
          fprintf(g.tellfile,"integer: %ld\n",i);
          BP_result=BP_integer(-1);
         }
      else
         BP_fail();
   }
 break;

 case 8: /* for experts: calling BinProlog from C */
   { cell L,R,Goal;

     BP_begin_put_list(L);
       BP_put_list(BP_string("one"));
       BP_put_list(BP_integer(2));
       BP_put_list(BP_string("three"));
       BP_put_list(BP_input); /* whatever comes as input */
     BP_end_put_list();

     BP_put_functor(Goal,"append",3);
     BP_put_old_var(L);
     BP_put_old_var(L);
     BP_put_new_var(R);

     BP_prolog_call(Goal); /* this will return NULL on failure !!!*/

     BP_put_functor(Goal,"write",1);
     BP_put_old_var(R);

     BP_prolog_call(Goal); /* calls write/1 */

     BP_put_functor(Goal,"nl",0);
```

```
      BP_prolog_call(Goal); /* calls nl/0 */

      BP_result=R; /* returns the appended list to Prolog */

   }
   break;

 case 10: /* for experts: calling BinProlog from C */
   { cell L,R,Goal;

     BP_begin_put_list(L);
       BP_put_list(BP_string("one"));
       BP_put_list(BP_integer(2));
       BP_put_list(BP_string("three"));
       BP_put_list(BP_input); /* whatever comes as input */
     BP_end_put_list();

     BP_put_functor(Goal,"cut_test",3); /* see c.pl */
     BP_put_old_var(L);
     BP_put_old_var(L);
     BP_put_new_var(R);

     BP_prolog_call(Goal); /* this will return NULL on failure !!!*/

     BP_put_functor(Goal,"write",1);
     BP_put_old_var(R);

     BP_prolog_call(Goal); /* calls write/1 */

     BP_put_functor(Goal,"nl",0);

     BP_prolog_call(Goal); /* calls nl/0 */

     BP_result=R; /* returns the appended list to Prolog */

   }
   break;

 case 11: /* CALLING BinProlog on a NEW ENGINE */
   { stack Engine;
     cell Goal,Answer;
     cell Xs,X;

     BP_begin_put_list(Xs);
       BP_put_list(BP_string("new_engine_test"));
       BP_put_list(BP_input); /* whatever comes as input */
       BP_put_list(BP_integer(100));
       BP_put_list(BP_integer(200));
       BP_put_list(BP_integer(300));
     BP_end_put_list();

     BP_put_functor(Goal,"member",2);
     BP_put_new_var(X);
     BP_put_old_var(Xs);
```

```c
        Engine=BP_create_engine(wam,100,50,50);

        BP_load_engine(Engine,Goal,X);

        while((Answer=(cell)ask_engine(Engine)))
          { cell Goal1;

            BP_put_functor(Goal1,"write",1);
            BP_put_old_var(Answer);

            BP_prolog_call(Goal1); /* calls write/1 */

            BP_put_functor(Goal1,"nl",0);

            BP_prolog_call(Goal1); /* calls nl/0 */

          }
        BP_destroy_engine(Engine);
        BP_result=Goal; /* returns the goal - just for tracing */
      }
      break;

  case 12: /* CALLING BinProlog on a NEW ENGINE. The friendliest way... */
      { stack Engine;
        cell Goal,Answer;
        cell Xs,X;

        BP_sread("[0,s(0),s(s(0)),[a,b,c]]",Xs);

        BP_put_functor(Goal,"member",2);
        BP_put_new_var(X);
        BP_put_old_var(Xs);

        Engine=BP_create_engine(wam,100,50,50);

        BP_load_engine(Engine,Goal,X);

        printf("BP_input: = %s\n",BP_swrite(BP_input));

        while((Answer=(cell)ask_engine(Engine)))
          {
            printf("Answer = %s\n",BP_swrite(Answer));
          }
        BP_destroy_engine(Engine);
        BP_result=Goal; /* returns the goal - just for tracing */
      }
      break;


  case 13: /* CALLING BinProlog on a NEW ENGINE. The friendliest way... */
      { stack Engine; int i;
        cell Goal,Answer,Answers[4],Ys;
        cell Xs,X;
```

```
    BP_sread("[0,s(0),s(s(0)),[a,b,c]]",Xs);

  BP_put_functor(Goal,"member",2);
  BP_put_new_var(X);
  BP_put_old_var(Xs);

  Engine=BP_create_engine(wam,100,50,50);

  BP_load_engine(Engine,Goal,X);

  /* accumulates the answers */

  for(i=0; Answer=(cell)ask_engine(Engine); i++)
      {
        /* To survive failure in Engine !!!!
           After copying, Answer will point to a valid object on
           the calling engine's heap whose H is also updated (see
           BP_copy_answer in c.h).
        */
        BP_copy_answer(Answer);
        Answers[i]=Answer;
      }
  BP_destroy_engine(Engine);

  /* prints out the accumulated answers */

  printf("BP_input: = %s\n",BP_swrite(BP_input));
  BP_begin_put_list(Ys);
  for(i=0; i<4; i++)
      {
        printf("Answers[%d] = %s\n",i,BP_swrite(Answers[i]));
        BP_put_list(Answers[i]); /* to return them to Prolog */
      }
  BP_end_put_list();
  BP_result=Ys; /* returns the list of answers, Ys,
                   after constructing it on the heap */
}
break;

/**
  Calls a string transformer function from BinProlog:
  sfun gets a char * argument which it modifies at will.
  Signals succes with 1, failure with 0
*/
case 14:
  BP_result=BP_funptr(sfun);
break;

/**
  Calls a C function with an int or float argument
  and returns a float result
*/
case 15: {
```

```
        double d;
        if(BP_is_integer(BP_input)) {
            int i;
            BP_get_integer(BP_input,i);
            d=(double)i;
        }
        else if(BP_is_atom(BP_input)) { // it means ref to a BP_FLOAT
            char *s=NULL;
            BP_get_string(BP_input,s);
            d=atof(s);
        }
        else if(BP_is_var(BP_input)) { // it means ref to a BP_FLOAT
            int ok;
            BP_get_float(BP_input,d,ok);
            if(!ok) {
              printf("not a float: = %s\n",BP_swrite(BP_input));
              BP_fail();
            }
        }
        else  /* this returns with failure */ {
            printf("BAD BP_input: = %s\n",BP_swrite(BP_input));
            BP_fail();
        }

        /* compute results */
        {  double r=ffun(d);
            BP_make_float(BP_result,r);
        }
    }
    break;

    /*
      getting any simple (int,double,string) data from
      prolog - the C side trusts that Prolog sends
      the right data - which isconverted
      automatically based on its dynamic Prolog type
    */
    case 16: {
     if(0)
      { string s=NULL;
        void *p=(void*)&s;
        BP_get_simple(BP_input,p);
        if(NULL==s) BP_fail();
        printf("got string = %s\n",s);
        BP_result=BP_atom(s);
      }
      else {
        double d;
        void *p=(void*)&d;
        BP_get_simple(BP_input,p);
        printf("got double = %f\n",d);
        BP_make_float(BP_result,d+1);
      }
    }
```

```
        break;

        /*
          getts a list of known length, applies
          to the a function of type int f(void **)
          of simple args and returns an int
          a list like [hello,3.14,2003] will
          need argc=0 and argv should have the address
          of variable of APPROPRIATE types. Conversion
          of data is automatic but we trust YOU to provide
          the right containers in C !!!
        */
        case 17: {
          string s;
          double d;
          int i,n;
          int argc=3;
          void *argv[3] ={(void*)&s,(void*)&d,(void*)&i};
          if(NULL==BP_get_list(BP_input,argc,argv)) BP_fail();
          printf("got from prolog list: %s, %f, %d\n",s,d,i);
          BP_result=BP_integer(i);
        }
        break;

       /* EDIT AND ADD YOUR CODE HERE....*/

     default:
      return LOCAL_ERR(X(1),"call to unknown user_defined C function");
  }
  return H;
}
```

1.3   Calling Prolog from C

Normally this is done after a first call from Prolog to C (this gives a chance to the prolog system to get initialized). The most flexible technique is based on multiple engines. Take a look at `case 11` and `case 12` in the previous section.

The lower-level interface, (which follows) is still usable but less recommended.

```
/* this can be used to call Prolog from C : see example if0 */

term bp_prolog_call(goal,regs,H,P,A,wam)
  register term goal,regs,H,*A;
  register instr P;
  register stack wam;
{
  PREP_CALL(goal);
  return bp(regs,H,P,A,wam);
}

/* simple example of prolog call */
term if0(regs,H,P,A,wam)
  register term regs,H,*A;
```

```
  register instr P;
  register stack wam;
{ term bp();
  cell goal=regs[1];

  /* in this example the input GOAL is in regs[1] */
  /* of course you can also build it directly in C */
  /* unless you want specific action on failure,
     use BP_prolog_call(goal) here */

  H=bp_prolog_call(goal,regs,H,P,A,wam);
  if(H)
     fprintf(stderr,"success: returning from New WAM\n");
  else
     fprintf(stderr,"fail: returning from New WAM\n");

  /* do not forget this !!! */
  return H; /* return NULL to signal failure */
}
```

BinProlog's `main()` should be the starting point of your program to be able to initialize all data areas. To call back from C you can follow the example if0. A sustained BinProlog-C dialog can be set up by using the 2 techniques described previously.

An example of using the C-interface (composed of files c.h c.c c.pl) can be found in directory `c_inter`. If you whish you can create a standalone C-executable by using BinProlog's compilation to C (see directories `pl2c`, `dynpl2c`) you can simply type 'make' in directory `c_inter`.

2 A 3-tier BinProlog/C/Java interface through JNI

Simple BinProlog Java interface for Windows, Solaris and Linux.

2.1 Overview

The BinProlog runtime emultor combined with the C-ified compiler are packaged into a dynamic library (jbp.dll or libjbp.so). A stub jBinPro.c based on BinProlog's C interface implements a call_bp_main C function which is declared as a native Java method in file JavaLog.java.

A makefile (you might whish to edit for path information) takes care of the whole process.

The scripts jbp.bat (Windows) and jbp (Solaris) can be used to run the resulting application. You might have to edit them for path information.

More work, BinProlog Professional source license and possibly the sources of the JDK kit from Sun are needed to make this into an applet runnable from Netscape. Note also that currently native methods in Java are of limited use due to security concerns and that such an applet might be restricted to your INTRANET.

I would suggest using BinProlog's Linda-based Jinni extension for unrestricted INTER-NET applications or a combination of Java and BinProlog components.

Still, for a minimal overhead interaction, in case your interface needs to glue together C/C++ Prolog and Java components, the effort might be worth it. On platforms where BinProlog does not support multi-threading, using Jinni's (a simplified Java based Prolog dialect) might be particularly appealing.

2.2 The Prolog Component

This example test bidirectional BinProlog/Java connection going through BinProlog's C interface. The Java application is Jinni, which also provides Prolog functionality. These can be seen also as an interface between a Prolog-in-Java and a Prolog-in-C.

```
main:-
  write('BinProlog called from Java'),nl,
  % WILL CALL Jinni services as follows:
  % ask_jinni(Pattern,Goal,Answer) Answer=the(Pattern') | no
  write('calling back Java using new_bultin, with:'),nl,
  In='listing(append)',
  write(In),nl,
  new_builtin(1,In,Out),
  write('back to Prolog again'),
  write(Out),nl,
  !.
main:-
  write('something failing in main.pro'),nl.
```

2.3 The Java Component

This is the master entry point of the application as BinProlog (written in C) is seen as a Java native method (using JNI, JDK 1.1 convetions). When started with `java jbp` its `main` method will load the native method obtained by linking jbp.c with BinProlog object files. Then it calls BinProlog's main entry C function with main.pro as parameter. It also provides a Java method `fromC` to be called back (this is supported by JNI 1.1) later.

```
class jbp {
  public native String call_bp_main(String jGoal);
  static {
    System.loadLibrary("jbp");
  }

  public  String fromC(String k) { // constructor

        System.out.println("back from java:"+k);

        return "CallFromCtoJava, "+k;
  }

  public static void main(String args[]) {
    jbp a_jbp=new jbp();
    String Answer=a_jbp.call_bp_main("$main.pro");
    System.out.println("ANSWER:"+Answer);
  }
}
```

2.4 The C component

The C component is a based simplified version of `new_builtin` from directory `c_inter`, showing calls from BinProlog to C and back.

To make some sense out of this, the reader has to start by understanding the JNI from the documentation at www.javasoft.com and paly first with our C-interface described in the previous section.

```c
#include <jni.h>
#include "jbp.h"
#include <stdio.h>

/* code BinProlog will call before after its own initalization process */

int init_c() {
  printf("initialising user's C code\n");
  return 1;
}

/* MAIN FILE: change this if you want to call BinProlog as a DLL */
/* or some other form of dynamically linked library            */

/***** BinProlog's embedding as a Java native procedure      *****/

/*
 * Class:      jbp
 * Method:     call_bp_main
 * Signature: (Ljava/lang/String;)V
 */

static JNIEnv *jenv=NULL;
static jobject jobj=NULL;

JNIEXPORT jstring JNICALL
Java_jbp_call_1bp_1main (JNIEnv *env, jobject obj, jstring jGoal) {
int argc=2; char *argv[]={"bp","$main.pro",NULL};
extern int bp_main(int argc, char **argv);
  argv[1]=(*env)->GetStringUTFChars(env, jGoal, 0);
  jenv=env;
  jobj=obj;
(void) bp_main(argc,argv);
  return jGoal;
}

/****************************************************************
          STUB IN CASE THE C_INTERFACE IS NOT USED

(Real) files using new C-interface (c.pl c.h c.c have been moved to
directory ../c_inter
****************************************************************/

#include "global.h"

extern struct specsyms g;
```

```
string c_interface="with Java and C Interface";

static string c_errmess="c.c: prototype Java interface\n";

term new_builtin(register term H, register term regs, register term *A,
                 register instr P, register stack wam)
{
  jstring jback,jto;   const char* cback;   const char* csend="FromCtoJ";
  jclass cls = (*jenv)->GetObjectClass(jenv, jobj);
  jmethodID mid = (*jenv)->GetMethodID(jenv, cls,"fromC",
  "(Ljava/lang/String;)Ljava/lang/String;"
  );

  fprintf(STD_err,c_errmess);


  if (mid == 0) return NULL;
  jto=(*jenv)->NewStringUTF(jenv, csend);
  jback=(*jenv)->CallObjectMethod(jenv, jobj, mid,jto);
  cback=(*jenv)->GetStringUTFChars(jenv, jback, 0);
  printf("BACK!%s\n",cback);

  /*return NULL;  always fails */
  regs[1]=regs[4];
  *H=regs[2];
  regs[3]=(cell)H++;
  return H;
}

term if0(register term regs, register term H, register instr P,
         register term *A, register stack wam)
{ /* unused builtin */
  fprintf(STD_err,c_errmess);
  return NULL;
}
```

3 BinProlog's Tcl/Tk interface (written in cooperation with Bart Demoen, KU. Leuven, back in 1993 or 1994)

This interface is based on and abstract scheme for the embedding of two languages with strong meta-programming, parsing and structure manipulation capabilities. We provide a call-back mechanism requiring each language to *compile* its service requests to the other's syntax for processing through the other's *meta-programming* capabilities. *State mirroring* ensures that only differential information needs to be transmitted. We describe a practical implementation of these principles and our experience with the **BinProlog Tcl/Tk** interface. Compilation by each side, that targets the other's syntax and dynamic object manipulation capabilities, achieves in *less than two hundred* lines of code, a fully functional integration of the two systems. The environment is portable to any Prolog system to which it adds powerful graphic and transparent distributed network programming facilities. Our

approach suggest that logic programming language-processors can be made to fit easily in a multi-paradigm programming environment with little programming effort by using their strong *language translation* and *meta-programming* capabilities.

3.1   Introduction

Integration of various programming paradigms is a necessity in modern programming environments. Monolithic language implementations tend to be replaced more and more by a combination of specialised *tools*. In particular, generic graphic processors can be connected to various languages which do not need their own graphic primitives anymore.

However, when there's a large semantic distance between the languages and they feature radically different computational models, a low-level interface is a painful and not always rewarding programming task. It happens very often that when the interface is finished one of the sides of the interface just becomes obsolete because of rapid evolution. This is especially true with various windowing systems which are often machine/operating system dependent.

The aim of this paper is to propose an unusually quick interfacing technique which uses meta-programming capabilities on the two sides. The basic idea is to link the two languages through a standard client-server interface and have each of them drive the other as an interactive agent, by generating the appropriate code on the fly. By mirroring the state of the objects, only state changes have to be transmitted from one side to the other, so that the granularity of the interaction will not become a bottleneck. Moreover, if one side has the capacity to react to events, this property will be inherited with minimal programming effort by the other side.

This paper is motivated by our work on finding a simple and portable way to add a visual programming environment to Prolog systems.

The popular Tcl/Tk visual language by John Ousterhout [**?**] is basically a composite made of of a shell-like interpreted programming language (Tcl) and a high-level Motif-style graphic package (Tk). Tcl is an untyped string-only language with strong meta-programming facilities (i.e. an *eval* primitive and dynamic procedure creation).

After describing a general language-processor interaction model we will report how it has been applied to an interface between BinProlog [**?**], and Tcl/Tk and how this interface has been ported to another Prolog system (Prolog by BIM) with minimal programming effort.

3.2   An abstract multi-language communication model

Let L and R be two languages, $t_{L,R}$ a translation function (compilation) from L to R, $t_{R,L}$ the reverse translation from R to L. Let e_L and respectively e_R be the *eval* operations of L and R. Let r_L, w_L, r_R and w_R denote the read and write operations of languages L and R.

An L_R-processor is a process with capabilities to

1. execute programs written in L

2. implement a translation function from L to R.

We say that two language processors L_R and R_L are *connected* if the following conditions are verified:

1. each *translate+write* operation on one side triggers exactly one *read+eval* operation on the other side,

2. one or more[2] *translate+write* operations can be triggered by each *eval* operation.

---

[2]The ability to trigger more than one *translate+write* is needed for instance when more than one task has to be initiated or more than on component of the state of one side has to be updated in a transaction.

This can be formalized more precisely in terms of temporal modal operators.

Our *write* and *read* operations should not be confused with ordinary input-output primitives. More precisely they are abstracted from the subset of input-output operations which can be meaningfully translated and are therefore suitable as input for the other side's eval functions. Thus they can be seen as carrying messages from an L_R processor to an R_L processor.

We say that an L_R and an R_L processor are *fairly interacting* if

1. each write operation on one side will be eventually read on the other side

2. the behaviour of each side is observable on the other side in terms of a sequence of successive read and write operations

If for two L_R and R_L language processors are *connected* and their read and write operations are queued, then they are *fairly interacting.*

We will break this fully symmetric behaviour by defining a *slave* language-processor as one that passively waits for write operations from the other side and a *master* language-processor which reacts to a write operation by a corresponding read operation with its result passed to its *eval* operation. We suppose also that the user interacts only with the master. As only the slave waits blocked on read operations, the master is free to be 'multi-threaded' or event-driven.

3.3   The BinProlog to Tcl/Tk interface

As an instance of this general scheme, we have chosen to connect BinProlog as a 'slave process' to a Tcl/Tk *wish* shell enhanced with reactive capabilities given by the *addinput* facility [3]. A specialized BinProlog toplevel is launched from the Tcl/Tk shell as a background Unix process connected through a bidirectional pipe. The BinProlog process is able to generate and react to Tcl/Tk events.

The interface uses tcl7.3 with tk3.6 combined with addinput-3.6a. It consists of 91 lines of Prolog and 101 lines of (commented) Tcl code. Thanks to the strong meta-programming capabilities of both languages the interface boldly *compiles* messages from one side to evaluable representations on the other side and calls the appropriate *eval* operation. The interface is portable to other Prolog systems and requires no C-programming. The installation of a BinProlog slave on the 'event channel' handled by `addinput` is done as follows:

```
proc start_prolog {} {
#   Opening a bidirectional pipe to BinProlog
#   under control of the addinput facility
    set f [open "|bp -q5 server.bp" r+]
    addinput $f "bpInOut %% %E %F"
}
```

Output from the BinProlog slave `bp server.bp` activates a Tcl/Tk procedure which executes on a *new* Tk-interpreter without disturbing normal interaction with the *wish* shell.

Basically, performance is not affected by the parsing as this is kept minimal. Moreover in systems with a parser written in C (as Prolog by BIM or the Koen De Bosschere's ISO-parser based version of BinProlog) parsing is in itself fast enough not to dominate the interaction

---

[3]A modification of the Tcl/Tk event loop (freely available from ftp.neosoft.com) which enables it to react to the presence of new input by triggering the execution of a user specified procedure when input becomes available.

cost. Techniques as *state mirroring* which will be discussed in the next section are also used to minimize the communication overhead.

High-level primitives on the Tcl/Tk side ensure that output to Prolog are valid Prolog terms. On the Prolog side a special toplevel loop reads the Tcl/Tk message, and applies its eval operation (call/1) to it. To avoid spurious interaction due to syntax errors or unexpected messages on each side only lines having a reserved header (i.e. `call_prolog` and respectively `call_tcl`) are evaluated. However, to give the look and feel of interacting with a Prolog system, messages from Prolog which are not of the form `call_tcl {...}` will be printed out as such. This fits well with Tcl which is string oriented but would not fit well with a term-oriented language like Prolog without parser modifications.

3.4   State mirroring

State mirroring is a simple technique that consists of duplication by a given language processor of (relevant) state information of an interactively connected other language processor.

In the case of Prolog, we have chosen to avoid the representation of states by infinite forward loops for two reasons:

1. unexpected failure would make the system unreliable and hard to debug

2. unexpected delays induced by garbage collection would make a visual interface unpleasant to use

Fortunately, BinProlog's blackboard turned out to be the perfect match to mirror the state of various Tcl/Tk objects. A library emulating BinProlog's blackboard operations in terms of generic Prolog dynamic database operations has been used to achieve the same effect in the Prolog by BIM port of the interface.

3.5   Programming with the embedded environment

Using the BinProlog-Tcl/Tk interface is very simple. Let `<P>` be the name of the composite program. Suppose the user creates 2 files `<P>.tcl` and `<P>.pl`. The composite 'program' can be executed as follows:

```
$ wish
% source <P>.tcl
```

`<P>.tcl` is expected to have at its end something like:

```
start_prolog
p {compile(<P>)}
```

The user will get the usual Tcl command prompt to interact with either Tcl or Prolog. A command entered at the Tcl prompt normally 'goes to' Tcl except for the following:

```
start_prolog      -connects to a new Prolog process
halt_prolog       -terminates the Prolog process:

p {<PrologTerm>}  -sends a goal to Prolog for quiet evaluation

q {<PrologTerm>}  -sends a query to Prolog and gets back answers
```

After being activated by one of the previous (p or q) Tcl commands a Prolog goal may 'call back' to continue the dialog with Tcl/Tk using the following Tcl command on the Prolog side:

```
call_tcl/1    -sends to Tcl a command to be immediately
              evaluated in 'background', while the Tcl shell
              prompts and waits for Tcl or Prolog commands.
```

Output with `puts` (Tcl) or `write/1` (Prolog) goes as usual to the `stdout` of the Tcl shell.

3.6   A visual N-queens program

This program illustrates two simple-minded but useful features of the interface:

1. Prolog lists are sent to Tcl as Tcl-lists

2. Prolog simply pumps solutions one by one using their most natural representation through `call_tcl(display_queens(L))`.

3. synchronization is done as follows:

   - the user provides mouse events generating write operations
   - the prolog process waits for input on a read operation until asked for another answer

   Equivalently, the user can also control the Prolog program from Tcl's command line by typing in "p more" or "p done" messages.

   The code on the Prolog side consists of a classic N-queens program, the generic prolog interface program server.pl and the clause:

```
qs(N):-
    queens(N,L),
    call_tcl(display_queens(L)),   % sends a display request
    write('type <p done> when finished <p more> otherwise'),nl,
    tcl_in(call_prolog(done)),!.        % waits for 'done' or 'more'
```

The code on the Tcl/Tk side consists of the generic server.tcl interface program, routines to set up the visual display and procedures like `show_queen`, `hide_queen` which are called by the main Tcl/Tk-side routine:

```
proc display_queens {qs} {
  global w count
  incr count
  hide_queens
  set l 0
  foreach q $qs {
    incr l
    show_queen $w [expr $l -1] [expr $q -1]
  }
}
```

This procedure simply counts the answers and updates the display when a new answer comes from Prolog, triggered by the presence of new input on the pipe.

3.7   Performance evaluation

Tcl/Tk is itself a glue language which links together large blocks of C-code. We have written a naive-reverse program and measured that it executes in Tcl between 20-50 times slower than in BinProlog. This definitely suggests that our technique, using relatively expensive parsing and compilation operations will not become itself an interaction bottleneck and therefore due to the much higher flexibility and almost instant portability between various Prolog systems this approach looks superior to a highly Prolog-specific C-level integration.

As BinProlog is a much faster symbolic processor than Tcl, we have decided to do most of the translation operations on the Prolog side. We have measured about 1500 BinProlog-Tcl/Tk exchanges per second on a Sparcstation 2 and 700-800 exchanges on a Sparcstation 1, both serving the same Tektronix X-terminal through a network.

This confirms that the interface is fast enough for complex real time visualisation tasks and pleasant to use in a average Unix environment.

3.8   Porting the interface to another Prolog

It took one hour to make the interface work with ProLog by BIM. The main change was to modify `start_prolog` on the Tcl side to start a ProLog by BIM process with:

```
set f [open "|bim server.pro" r+]
```

The ProLog by BIM process is initialized from a small new toplevel `server.pro`. The one hour work had more to deal with the differences between the two Prolog systems, than the interface itself. Moreover, code written for one Prolog system (like the `N-queens` program) runs without changes on the other, as far as it is written in portable Prolog. We think this compares favorably with a dedicated piecewise interface written in C for each Prolog graphic builtin in terms of programming effort, portability and learning curve for the user, and, as our performance evaluation has shown, with reasonable real-time interactivity.

3.9   Related work

Another Prolog with Tcl/Tk interface we know of is that developed by Micha Meier at ECRC as the standard Eclipse GUI. It has been ported to SICStus Prolog as well. Micha Meier's interface is tightly integrated (at C-level) and Eclipse specific (although a Sicstus port is also provided). The tighter integration is better (in principle) in the case of a very low granularity communication, although the programming effort to implement it is more considerable than ours. Our performance analysis shows that global costs are dominated by process switching, graphic manipulations and brute force processing in Prolog. Moreover, our interface, does not rely on a specific Tcl/Tk implementation and has no modification to the C-code neither on the Prolog emulator neither on the Tcl/Tk side. Because of this reasons, we believe that the simplicity and portability of our design is a clear advantage, especially in the context of rapidly evolving backward incompatible Tcl/Tk releases.

4   Projected future work on the interface

We plan to make the interface fully Prolog-independent and to write Prolog libraries to support various existing Tcl/Tk extensions and tools for distributed Tcl/Tk programming, automatic interface generators etc. A Tcl/Tk based generic Prolog toplevel would also be very helpful to give the full illusion of *being in Prolog* while benefiting of visual goodies of a Tcl/Tk shell.

4.1 Conclusion on the interface

We have outlined a generic programming framework for a seamless integration of two programming languages which have 'eval' capability and implemented an instance of the framework as a Prolog to TCL/Tk interface.

Our experiments show that support for metaprogramming tools in modern programming environment is important not only for the expressiveness of the language itself but also multiparadigm language integration.

We have shown that with a reduced programming effort a powerful interface can be built between a Prolog system and a state-of-the art visual environment.

This suggests that embedding of a logic engine in a generic multi-language environment is competitive in functionality and performance with the traditional approach which advocates to hardwire in the Prolog system a set of language-specific extensions.

The code for the interface can be obtained by ftp from **clement.info.umoncton.ca** and has been integrated in the BinProlog starting with the 2.20 distribution.

5   Interfacing with Internet Languages

Our approach to multi-language/multi-machine communication through the net has tried to keep the same touch of simplicity as the previously described Tcl/Tk interface [**?**]. Instead of exposing low-level C operations, we have chosen to build more convenient abstractions.

Related BinProlog documentation is available at: [**?**, **?**, **?**, **?**].

Our recent Jinni and BinProlog related papers are available from:

```
http://www.cs.unt.edu/~tarau/
```

Related publications: [**?**, **?**, **?**, **?**]