WWW Programming using Computational Logic Systems (and the PiLLoW/CIAO Library)*

Daniel Cabeza and Manuel Hermenegildo
{dcabeza,herme}@dia.fi.upm.es

CLIP Group
Computer Science Department
Technical University of Madrid (UPM), Spain

Abstract

We discuss from a practical point of view a number of issues involved in writing Internet and WWW applications using LP/CLP systems. We describe PiLLoW, a public-domain Internet and WWW programming library for LP/CLP systems which we argue significantly simplifies the process of writing such applications. PiLLoW provides facilities for generating HTML structured documents, producing HTML forms, writing form handlers, accessing and parsing WWW documents, and accessing code posted at HTTP addresses. We also describe the architecture of some application classes, using a high-level model of client-server interaction, *active modules*. We then propose an architecture for automatic LP/CLP code downloading for local execution, using generic browsers. Finally, we also provide an overview of related work on the topic. The PiLLoW library has been developed in the context of the &-Prolog and CIAO systems, but it has been adapted to a number of popular LP/CLP systems, supporting most of its functionality.
Keywords: WWW, HTML, HTTP, Distributed Execution, (Constraint) Logic Programming

1 Introduction

The wide diffusion of the Internet and the popularity of WWW ("World Wide Web" [1]) protocols are effectively providing a novel platform that facilitates a new class of highly sophisticated distributed applications. Good support for network connectivity and the protocols and communication architectures of this novel platform are obviously requirements for any programming tool to be useful in this arena. However, this alone may not be enough. It seems natural that significant parts of network applications will require symbolic and numeric capabilities which are not necessarily related with distribution. Important such capabilities are, for example, high-level symbolic information processing, dealing with combinatorial problems, and natural language processing in general. Logic Programming (LP) [22, 10] and Constraint Logic Programming (CLP) systems [20, 13, 11, 24, 12] have been shown particularly successful at tackling these issues (see, for example, the proceedings of recent conferences on the "Practical Applications of Prolog" and "Practical Applications of Constraint Technology"). It seems natural to study how LP/CLP technology fares in developing applications which have to operate over the Internet.

In fact, Prolog, its concurrent and constraint based extensions, and logic programming languages in general have many characteristics which appear to set them particularly well placed for making an impact on the development of practical networked applications, ranging from the simple to the quite sophisticated. Notably, LP/CLP systems share many characteristics with other recently proposed network programming tools, such as Java, including dynamic memory management, well-behaved structure and pointer manipulation, robustness, and compilation to architecture-independent bytecode. Furthermore, and unlike the scripting or application languages currently being proposed (e.g., shell scripts, Perl, Java, etc.), LP/CLP systems offer a quite unique set of additional features including dynamic databases, search facilities, grammars, sophisticated meta-programming, and well understood semantics.

*This paper is an improved version of [7].

In addition, most LP/CLP systems also already offer some kind of low level support for remote communication using Internet protocols. This generally involves providing a *sockets* (or ports) interface whereby it is possible to make remote data connections via the Internet's native protocol, TCP/IP. A few systems support higher level functionality layers on top of this interface including linda-style blackboards (e.g., SICStus Prolog [8] and &-Prolog/CIAO[1][17, 18, 15], BinProlog/μ^2-Prolog [29, 2], etc.) or shared variable based communication (e.g., KL1 [9], AKL [21], Oz [26], &-Prolog/CIAO [16, 4], etc.). In some cases, this functionality is provided via libraries, building on top of the basic TCP/IP primitives. This is the case, for example, of the SICStus (and CIAO) linda interface. In fact, as we have shown, shared variable based communication can also be implemented in conventional systems via library predicates, by using attributed variables [16, 4].

WWW applications generally use higher level protocols (such as `HTTP` or `FTP`) and application architectures (e.g., the cgi-bin interface) which are different from the shared variable or linda-based protocols. In this paper we study how good support for such protocols and architectures can be provided for LP/CLP systems, building on the basic, widely available TCP/IP protocols. Our aim is to discuss from a practical point of view a number of the new issues involved in writing Internet and WWW applications using LP/CLP systems, as well as the architecture of some typical applications. In the process, we will describe P*i*LL*o*W ("Programming in Logic Languages on the Web"), a public domain Internet/WWW programming library for LP/CLP systems which, we argue, significantly simplifies the process of writing such applications. P*i*LL*o*W provides facilities for generating HTML structured documents by handling them as Herbrand terms, producing HTML *forms*, writing form handlers, accessing and parsing WWW documents, and accessing code posted at HTTP addresses. We also describe the architecture of some relatively sophisticated application classes, using a high-level model of client-server interaction, *active modules* [4]. Finally we describe an architecture for automatic LP/CLP code downloading for local execution, using just the library and generic browsers.

The argument throughout the paper is that, with only very small limitations in functionality (which disappear when concurrency is added to the system, as in systems such as BinProlog/μ^2-Prolog, AKL, Oz, KL1, and &-Prolog/CIAO), it is possible to add an extremely useful Internet/WWW programming layer to any LP/CLP system without making any significant changes in the implementation. We argue that this layer can simplify the generation of applications in LP/CLP systems including active WWW pages, search tools, content analyzers, indexers, software demonstrators, collaborative work systems, MUDs and MOOs, code distributors, etc.

The P*i*LL*o*W library has been developed in the context of the &-Prolog and CIAO systems, but it has been adapted to a number of popular LP/CLP systems, supporting most of its functionality. This document can serve also as a WWW/HTML primer, containing sufficient information for developing relatively high-complexity WWW applications in Prolog and other LP and CLP languages.

2 Writing basic cgi-bin applications

The simplest way of writing WWW applications is through the use of the "Common Gateway Interface" (CGI). A CGI executable is a standard executable file but such that the HTTP server can tell it in fact contains a program that is to be run, rather than a document text that is to be sent to the client as usual. The file can be distinguished by belonging to a special directory, commonly named `cgi-bin`, or by a special filename ending, such as `.cgi`. The basic idea behind this interface is illustrated in Figure 1. When the user selects an address of a CGI executable in a document, such as `http://www.xxx.yyy/cgi-bin/hello_world` (or perhaps `http://www.xxx.yyy/foo/hello_world.cgi`) the browser issues a standard document request (1). The `HTTP` server, recognizing that it is a CGI executable rather than a document, starts the executable (2), and during such execution stores the output of the executable in a buffer (3). Upon termination of the executable, the contents of the buffer (which should be in a format that the browser can handle, such as HTML) are returned to the browser as if a normal page with that content had been accessed (4).

The following is an example of how a very simple such executable can be written in an LP/CLP language. The source might be as follows:[2]

```
main :-
```

[1] &-Prolog is an and-parallel version of Prolog. The &-Prolog extensions to Prolog (basically, the addition of high-level concurrency and communication operators) allow exploitation of parallelism (a parallelizing compiler is provided with the system) and basic forms of concurrent programming. CIAO is a set of libraries built on top of the &-Prolog kernel which allow sophisticated concurrent and distributed programming, constraint solving, support for several computation rules, etc.

[2] Note that in the examples presented the HTML may be slightly simplified, and thus may not be completely standard (however, the examples can be used as is with all popular browsers).

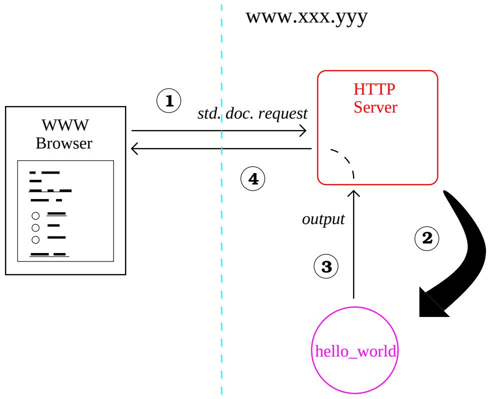

Figure 1: The CGI interface

```
    write('Content-type: text/html'), nl, nl,
    write('<HTML>'),
    write('Hello world.'),
    write('</HTML>').
```

And the actual executable could be generated as a saved state at the system prompt in the standard way. E.g., for most Edinburgh-style systems:

```
?:- compile('hello_world.pl'),
     save('/usr/local/etc/htppd/cgi-bin/hello_world'), main.
```

The address of the executable in machine `www.xxx.yyy` would then be

`http://www.xxx.yyy/cgi-bin/hello_world`.

In some systems, saved states have the disadvantage of their generally large size, but many systems have other ways of producing reasonably-sized executables. For example, in the &-Prolog/CIAO system compiled executables can be generated which are generally of smaller size than the source program.

3 LP/CLP Scripts for CGI Applications

CGI executables are often small- to medium-sized programs that do relatively simple tasks. This, added to the slow speed of the network connection in comparison with that of executing a program (which makes program execution speed less important) has made scripting languages (such as shell scripts or Perl) very popular for writing these scripts. The popularity is due to the fact that no compilation is necessary (extensive string handling capabilities also play an important role in the case of Perl), and thus changes and updates to the program imply only editing the source file.

Logic languages are, a priori, excellent candidates to be used as scripting languages[3]. However, the relative complication in making executables (needing in most systems to start the system, compile or consult the file, and make a saved state) and the often large size of the resulting executables may deter CGI application programmers. It appears convenient to provide a means for LP/CLP programs to be executable as scripts, even if with reduced performance.

It is generally relatively easy to support scripts with the same functionality in most LP/CLP systems (see [14] for an example developed for the CIAO system and adapted to SICStus). Let's assume that `lpshell` is a version of the LP/CLP system (for example, a saved state), which first loads the file given to it as the first argument (but excluding the first line and routing all loading messages away from the standard output) and then starts execution at `main/1` (the argument provides the list of command line options). Then, for example, in a Unix system, the following program can be run directly as a script without any need for compilation:

[3]For example, the built-in grammars and databases greatly simplify many typical script-based applications. Note that grammars are much more powerful than regular expressions, often found in other scripting languages, which in general can only provide an approximation to the solution.

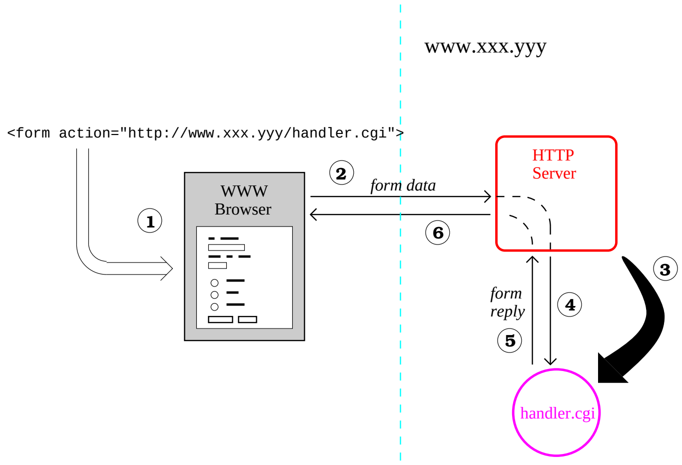

Figure 2: The Forms interface

```
#!/usr/local/bin/lpshell

main(_) :-
    write('Content-type: text/html'), nl,
    write('<HTML>'),
    write('Hello world.'),
    write('</HTML>').
```

4 Form Handling in HTTP

Since CGI executables only produce output, and this output is not a function of the input, CGI executables by themselves are only of limited interest. However, they become most useful when combined with HTML forms. HTML forms are HTML documents (or parts of HTML documents) which include special fields such as text areas, menus, radio buttons, etc. The steps involved in the handling of the input contained in a form are illustrated in Figure 2. When a document containing a form is accessed via a form-capable browser (Mosaic, Netscape, Lynx, etc.), the browser displays the input fields, buttons, menus, etc. indicated in the document, and *locally* allows the user to perform input by modifying such fields. However, this input is not ultimately handled by the browser. Instead, it will be sent to a "handler" program, which can be anywhere on the net, and whose address must be given in the form itself (1). Forms generally have a "submit" button such that, when pressed, the input provided through the menus, text areas, etc. is sent by the browser to the `HTTP` server corresponding to the handler (2). Two methods for sending this input exist: "GET" and "POST". In the meantime, the sending browser waits for a response from that program, which should come in the form of a new HTML document. The handler program is invoked in much the same way as a cgi-bin application (3), except that the information from the form is supplied to the handler (in different ways depending on the system, the method of invocation and the content type) (4). This information is encoded in a predefined format, which relates each piece of information to the corresponding field in the form, by means of a keyword associated with each field. The handler then identifies the information corresponding to each field in the original form, processes it, and then responds by writing an HTML document to its standard output (5), which is forwarded by the server to the waiting browser when the handler terminates (6). An important point to be noted is that, as with simple cgi-bin applications, the handler is started and should terminate for each transaction. The reader is referred, for example, to `http://kuhttp.cc.ukans.edu/info/forms/forms-intro.html` for a more complete introduction to CGI scripts and HTML forms.

5 Writing Form Handlers with PiLLoW

The PiLLoW library provides some predicates which simplify the task of getting the input from a form. As we said before, this input can be provided in several ways, depending on the system and the method used to invoke the form, and is encoded with escape sequences. The principal predicates provided are:

- `get_form_input(`*Dic*`)` Translates input from the form (with either the POST or GET methods, or ENCTYPE multipart/form-data) to a dictionary *Dic* of *attribute*=*value* pairs. It translates empty *value*s (which indicate only the presence of an attribute) to `'$empty'`, values with more than one line (from text areas or files) to a list of lines as strings, the rest to atoms or numbers. This is implemented using DCG parsers.
- `get_form_value(`*Dic*`,` *Var*`,` *Val*`)` Gets value *Val* for attribute *Var* in dictionary *Dic*. Does not fail: value is `''` if not found (this simplifies merging form producers and form handlers, see later).
- `text_lines(`*Val*`,`*Lines*`)` Transforms a value given by a dictionary to a list of lines, for data coming from a text area.
- `form_empty_value(`*V*`)` Useful to check that a value *V* from a text area is empty (can have spaces, newlines and linefeeds).
- `form_default(`*Val*`,`*Default*`,`*NewVal*`)` Useful when a form is only partially filled (and also in the first invocation of a combined form handler/producer – see Section 7). If the value of *Val* is empty then *NewVal*=*Default*, else *NewVal*=*Val*.
- `my_url(`*URL*`)` Returns in *URL* the Uniform Resource Locator (WWW address) of this cgi executable.
- `form_request_method((Method))` Returns in (Method) the method of invocation of the form handler ("GET" or "POST").

For example, suppose we want to make a handler which implements a database of telephone numbers and is queried by a form including a single entry field with name `person_name`. The handler might be coded as follows:

```
#!/usr/local/bin/lpshell

:- use_module('/usr/local/src/pillow/pillow.pl').

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    write('Content-type: text/html'), nl, nl,
    write('<HTML><TITLE>Telephone database</TITLE>'), nl,
    write('<IMG SRC="phone.gif">'),
    write('<H2>Telephone database</H2><HR>'),
    write_info(Name),
    write('</HTML>').

write_info(Name) :-
    form_empty_value(Name) ->
       write('You have to provide a name.')
  ; phone(Name, Phone) ->
       write('Telephone number of <B>'),
       write(Name),
       write('</B>: '),
       write(Phone)
  ; write('No telephone number available for <B>'),
    write(Name),
    write('</B>.').
```

```
phone(daniel, '336-7448').
phone(manuel, '336-7435').
phone(sacha,  '543-5316').
```

The code above is quite simple. On the other hand, the interspersion throughout the text of calls to `write` with HTML markup inside makes the code somewhat inelegant. Also, there is no separation between computation and input/output, as is normally desirable. It would be much preferable to have an encoding of HTML code as Prolog terms, which could then be manipulated easily in a more elegant way, and a predicate to translate such terms to HTML for output. This facility, provided by the PiLLoW library, is presented in the next section.

6 Handling HTML as Prolog Terms

Since LP/CLP systems perform symbolic processing using Herbrand terms, it seems natural to be able to handle HTML code directly as terms. Then, such structures only need to be translated by appropriate predicates to HTML code when they need to be output. In general, this relationship between HTML code and Prolog terms allows viewing any WWW page as a Herbrand term. The predicates which provide this functionality in PiLLoW are:

- `output_html(`*F*`)` Accepts in *F* an HTML term (or a list of HTML terms) and sends to the standard output the text which is the rendering of the term(s) in HTML format.
- `html2terms(`*Chars*`,`*Terms*`)` Relates a list of HTML terms and a list of ASCII characters which are the rendering of the terms in HTML format. This predicate is reversible (see later), `output_html/2` uses it to transform HTML terms in characters. Again, this is implemented via DCG parsing.

In an *HTML term* certain atoms and structures represent special functionality at the HTML level. An HTML term can be recursively a list of HTML terms. The following are legal HTML terms:

```
hello
[hello, world]
["This is an ", em('HTML'), " term"]
```

When converting HTML terms to characters, `html2terms/2` translates special structures into the corresponding format in HTML, applying itself recursively to their arguments. Strings are always leaved unchanged. HTML terms may contain logic variables, provided they are instantiated before the term is translated or output. This allows creating documents piecemeal, backpatching of references in documents, etc.

In the following sections we list the meaning of the principal Prolog structures that represent special functionality at the HTML level. Only special atoms are translated, the rest are assumed to be normal text and will be passed through to the HTML document.

6.1 General Structures

Basically, HTML has two kinds of components: HTML *elements* and HTML *environments*. An HTML element has the form "`<NAME` *Attributes* `>`" were `NAME` is the name of the element and *Attributes* is a (possibly empty) sequence of attributes, each of them being either an attribute name or an attribute assignment as `name="Value"`.

An HTML environment has the form "`<NAME` *Attributes* `>` *Text* `</NAME>`" were `NAME` is the name of the environment an *Attributes* has the same form as before.

The general Prolog structures that represent these two HTML constructions are:

- *Name*`$`*Atts* ('`$/2`' is defined as an infix, binary operator.) Represents an HTML element of name *Name* and attributes *Atts*, were *Atts* is a (possibly empty) list of attributes, each of them being either an atom or a structure *name*=*value*. For example, the term

  ```
  img$[src='images/map.gif',alt="A map",ismap]
  ```

  is translated into the HTML source

```
<img src="images/map.gif" alt="A map" ismap>
```

Note that HTML is not case-sensitive, so we can use lower-case atoms.

- *name*(*Text*) (A term with functor *name*/1 and argument *Text*) Represents an HTML environment of name *name* and included text *Text*. For example, the term

  ```
  address('clip@dia.fi.upm.es')
  ```

  is translated into the HTML source

  ```
  <address>clip@dia.fi.upm.es</address>
  ```

- *name*(*Atts*, *Text*) (This is a term with functor *name*/2 and arguments *Atts* and *Text*) Represents an HTML environment of name *name*, attributes *Atts* and included text *Text*. For example, the term

  ```
  a([href='http://www.clip.dia.fi.upm.es/'],"Clip home")
  ```

  represents the HTML source

  ```
  <a href="http://www.clip.dia.fi.upm.es/">Clip home</a>
  ```

- `env`(*Name*, *Atts*, *Text*) Equivalent to *Name*(*Atts*, *Text*).
- `begin`(*Tag*, *Atts*) It translates to the start of an HTML environment of name *Tag* and attributes *Atts*. There exists also a `begin`(*Tag*) structure. Useful, in conjunction with the next structure, when including in a document output generated by an existing piece of code (e.g. *Tag* = `pre`). Its use is otherwise discouraged.
- `end`(*Tag*) Translates to the end of an HTML environment of name *Tag*.

Now we can rewrite the previous example as follows:

```
#!/usr/local/bin/lpshell

:- use_module('/usr/local/src/pillow/pillow.pl').

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    output_html([
        'Content-type: text/html',
        html([title('Telephone database'),
              img$[src='phone.gif'],
              h2('Telephone database'),
              hr$[],
              Response)]).

response(Name, Response) :-
    form_empty_value(Name) ->
       Response = 'You have to provide a name.'
  ; phone(Name, Phone) ->
       Response = ['Telephone number of ',b(Name),': ',Phone]
  ; Response = ['No telephone number available for ',b(Name),'.'].

phone(daniel, '336-7448').
phone(manuel, '336-7435').
phone(sacha,  '543-5316').
```

Any HTML construction can be represented with these structures (except comments and declarations, which could be included as atoms or strings), but the PiLLoW library provides additional, specific structures to simplify HTML creation.

6.2 Specific Structures

In this section we will list the special structures for HTML which PiLLoW understands. A predicate `html_expansion/2` is provided to define new structures.

- `start` Used at the beginning of a document (translates to `<html>`).
- `end` Used at the end of a document (translates to `</html>`).
- `--` Produces a horizontal rule (translates to `<hr>`).
- `\\` Produces a line break (translates to `<br>`).
- `$` Produces a paragraph break (translates to `<p>`).
- `comment(`*Comment*`)` Used to insert an HTML comment (translates to `<!--` *Comment* `-->`).
- `declare(`*Decl*`)` Used to insert an HTML declaration – seldom used (translates to `<!`*Decl*`>`).
- `image(`*Addr*`)` Used to include an image of address (URL) *Addr* (translates to an `<img>` element).
- `image(`*Addr*`,`*Atts*`)` As above with the list of attributes *Atts*.
- `ref(`*Addr*`,`*Text*`)` Produces a hypertext link, *Addr* is the URL of the referenced resource, *Text* is the text of the reference (translates to `<a href="`*Addr*`">`*Text*`</a>`).
- `label(`*Label*`,`*Text*`)` Labels *Text* as a target destination with label *Label* (translates to `<a name="`*Label*`">`*Text*`</a>`).
- `heading(`*N*`,`*Text*`)` Produces a heading of level *N* ($1 \leq N \leq 6$), *Text* is the text to be used as heading – useful when one wants a heading level relative to another heading (translates to a `<h`*N*`>` environment).
- `itemize(`*Items*`)` Produces a list of bulleted items, *Items* is a list of corresponding HTML terms (translates to a `<ul>` environment).
- `enumerate(`*Items*`)` Produces a list of numbered items, *Items* is a list of corresponding HTML terms (translates to an `<ol>` environment).
- `description(`*Defs*`)` Produces a list of defined items, *Defs* is a list whose elements are definitions, each of them being a Prolog sequence (composed by `','/2` operators). The last element of the sequence is the definition, the other (if any) are the defined terms (translates to an `<dl>` environment).
- `nice_itemize(`*Img*`,`*Items*`)` Produces a list of bulleted items, using the image *Img* as bullet. The predicate `icon_address/2` provides a colored bullet.
- `preformatted(`*Text*`)` Used to include preformatted text, *Text* is a list of HTML terms, each element of the list being a line of the resulting document (translates to a `<pre>` environment).
- `entity(`*Name*`)` Includes the entity of name *Name* (ISO-8859-1 special character).
- `verbatim(`*Text*`)` Used to include text verbatim, special HTML characters (`<,>,&,"`) are translated into its quoted HTML equivalent.
- `nl` Used to include a newline in the HTML source (just to improve human readability).
- `cgi_reply` This is not HTML, rather, the CGI protocol requires this content descriptor to be used by CGI executables (including form handlers) when replying (translates to "`Content-type: text/html`").
- `pr` Includes in the page a graphical logo with the message "Developed using the PiLLoW Web programming library", which points to the manual and library source.

With these additional structures, we can rewrite the previous example as follows (note that in this particular example the use of `heading/2` or `h2/1` is equally suitable):

```
#!/usr/local/bin/lpshell

:- use_module('/usr/local/src/pillow/pillow.pl').

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    output_html([
        cgi_reply,
        start,
        title('Telephone database'),
        image('phone.gif'),
        heading(2,'Telephone database'),
        --,
        Response,
        end]).

response(Name, Response) :-
    form_empty_value(Name) ->
       Response = 'You have to provide a name.'
  ; phone(Name, Phone) ->
       Response = ['Telephone number of ',b(Name),': ',Phone]
  ; Response = ['No telephone number available for ',b(Name),'.'].

phone(daniel, '336-7448').
phone(manuel, '336-7435').
phone(sacha,  '543-5316').
```

We have not included above the specific structures for creating forms, they are included and explained in the following section.

6.3 Specific Structures for Forms

In this section we explain the structures which represent the various input elements that provide forms.

- `start_form`(*Addr*[,*Atts*]) Specifies the beginning of a form. *Addr* is the address (URL) of the program that will handle the form, and *Atts* other attributes of the form, as the method used to invoke it. If *Atts* is not present the method defaults to POST. (Translates to `<form action="`*Addr*`"` *Atts* `>`.)
- `start_form` Specifies the beginning of a form without assigning address to the handler, so that the form handler will be the cgi-bin executable producing the form.
- `end_form` Specifies the end of a form (translates to `</form>`).
- `checkbox`(*Name*,*State*) Specifies an input of type `checkbox` with name *Name*, *State*=on if the checkbox is initially checked (translates to an `<input>` element).
- `radio`(*Name*,*Value*,*Selected*) Specifies an input of type `radio` with name *Name* (several radio buttons which are interlocked must share their name), *Value* is the the value returned by the button, if *Selected*=*Value* the button is initially checked (translates to an `<input>` element).
- `input`(*Type*,*Atts*) Specifies an input of type *Type* with a list of attributes *Atts*. Possible values of *Type* are `text`, `hidden`, `submit`, `reset`, ... (translates to an `<input>` element).
- `textinput`(*Name*,*Atts*,*Text*) Specifies an input text area of name *Name*. *Text* provides the default text to be shown in the area, *Atts* a list of attributes (translates to a `<textarea>` environment).

- `menu(`*Name*`,`*Atts*`,`*Items*`)` Specifies a menu of name *Name*, list of attributes *Atts* and list of options *Items*. The elements of the list *Items* are marked with the prefix operator '`$`' to indicate that they are selected (translates to a `<select>` environment).

For example, in order to generate a form suitable for sending input to the previously described phone database handler one could type at a Prolog prompt:

```
?:- ['/usr/local/src/pillow/pillow.pl'],
    output_html([
        start,
        title('Telephone database'),
        heading(2,'Telephone database'),
        $,
        start_form('http://www.clip.dia.fi.upm.es/cgi-bin/phone_db.pl'),
        'Click here, enter name of clip member, and press Return:',
        \\,
        input(text,[name=person_name,size=20]),
        end_form,
        end]).
```

Of course, one could have also simply written directly the resulting HTML document:

```
<html>
<title>Telephone database</title>
<h2>Telephone database</h2>
<p>
<form method="POST"
 action="http://www.clip.dia.fi.upm.es/cgi-bin/phone_db.pl">
Click here, enter name of clip member, and press Return:
<br>
<input type="text" name="person_name" size="20">
</form>
</html>
```

7 Merging the Form Producer and the Handler

An interesting practice when producing HTML forms and handlers is to merge the operation of the form producer and the handler into the same program. The idea is to produce a generalized handler which receives the form input, parses it, computes the answer, and produces a new document which contains the answer to the input, as well as a new form. A special case must be made for the first invocation, in which the input would be empty, and then only the form should be generated. The following is an example which merges the producer and the handler for the phones database:

```
#!/usr/local/bin/lpshell

:- use_module('/usr/local/src/pillow/pillow.pl').

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    output_html([
        cgi_reply,
        start,
        title('Telephone database'),
        image('phone.gif'),
        heading(2,'Telephone database'),
```

```
        --,
        Response,
        start_form,
        'Click here, enter name of clip member, and press Return:',
        \\,
        input(text,[name=person_name,size=20]),
        end_form,
        end]).

response(Name, Response) :-
    form_empty_value(Name) ->
       Response = []
  ; phone(Name, Phone) ->
       Response = ['Telephone number of ',b(Name),': ',Phone,$]
  ; Response = ['No telephone number available for ',b(Name),'.',$].

phone(daniel, '336-7448').
phone(manuel, '336-7435').
phone(sacha,  '543-5316').
```

This combination of the form producer and the handler allows producing applications that give the impression of being interactive, even if each step involves starting and running the handler to completion. Note that forms can contain fields which are not displayed and are passed as input to the next invocation of the handler. This allows passing state from one invocation of the handler to the next one.

8 Accessing WWW documents

The facilities presented in the previous sections allow generating HTML documents, including forms, and handling the input coming from forms. In many applications such as search tools, content analyzers, etc., it is also desirable to be able to access documents on the Internet. Such access is generally done through protocols such as `FTP` and `HTTP` which are built on top of TCP/IP. In LP/CLP systems which have TCP/IP connectivity (i.e., a sockets/ports interface) the required protocols can be easily coded in the source language using such facilities and DCG parsers. At present, only the HTTP protocol is supported by P*i*LL*o*W. As with HTML code, the library uses an internal representation of Uniform Resource Locators (URLs), and provides predicates which translate between the internal representation and the textual form. The facilities provided by P*i*LL*o*W for accessing WWW documents include the following predicates:

- `url_info(`*URL*`,`*Info*`)` Translates a URL *URL* to an internal structure *Info* which details its various components and vice-versa. For now non-HTTP URLs make the predicate fail. E.g.
  `url_info('http://www.foo.com/bar/scooby.txt',Info)`
  gives `Info = http('www.foo.com',80,"/bar/scooby.txt")`,
  `url_info(URL, http('www.foo.com',2000,"/bar/scooby.txt")`
  gives `URL = "http://www.foo.com:2000/bar/scooby.txt"` (a string).
- `url_info_relative(`*URL*`,`*BaseInfo*`,`*Info*`)` Translates a relative URL *URL* which appears in the HTML page refered to by *BaseInfo* (given as an `url_info` structure) to a complete `url_info` structure *Info*. Absolute URLs are translated as with the previous predicate. E.g.
  `url_info_relative("/guu/intro.html", http('www.foo.com',80,"/bar/scoob.html"), Info)`
  gives `Info = http('www.foo.com',80,"/guu/intro.html")`
  `url_info_relative("dadu.html", http('www.foo.com',80,"/bar/scoob.html"), Info)`
  gives `Info = http('www.foo.com',80,"/bar/dadu.html")`.
- `url_query(`*Dic*`,`*Args*`)` Translates a list of *attribute*=*value* pairs *Dic* (in the same form as the dictionary returned by `get_form_input/1`) to a string *Args* for appending to a URL pointing to a form handler.
- `fetch_url(`*URL*`,`*Request*`,`*Response*`)` Fetches a document from the Internet. *URL* is the Uniform Resource Locator of the document, given as a `url_info` structure. *Request* is a list of options which

specify the parameters of the request, *Response* is a list which includes the parameters of the response. The request parameters available are:

`head` To specify that we are only interested in the header.

`timeout(`*Time*`)` *Time* specifies the maximum period of time (in seconds) to wait for a response. The predicate fails on timeout.

`if_modified_since(`*Date*`)` Get document only if newer than *Date*. An example of a structure that represents a date is `date('Tuesday',15,'January',1985,'06:14:02')`.

`user_agent(`*Name*`)` Provide a user-agent field.

`authorization(`*Scheme*`,`*Params*`)` Provides an authentication field when accessing restricted sites.

name`(`*Param*`)` Any other functor translates to a field of the same name (e.g. `from('user@machine')`).

The parameters wich can be returned in the response list include (see the HTTP/1.0 definition for more information):

`content(`*Content*`)` Returns in *Content* the actual document text, as a list of characters.

`status(`*Type*`,`*Code*`,`*Phrase*`)` Gives the status of the response. *Type* can be any of `informational`, `success`, `redirection`, `request_error`, `server_error` or `extension_code`, *Code* is the status code and *Phrase* is a textual explanation of the status.

`pragma(`*Data*`)` Miscellaneous data.

`message_date(`*Date*`)` The time at which the message was sent.

`location(`*URL*`)` Where has moved the document.

`http_server(`*Server*`)` Identifies the server responding.

`allow(`*methods*`)` List of methods allowed by the server.

`last_modified(`*Date*`)` Date/time at which the sender believes the resource was last modified.

`expires(`*Date*`)` Date/time after which the entity should be considered stale.

`content_type(`*Type*`,`*Subtype*`,`*Params*`)` Returns the MIME type/subtype of the document.

`content_encoding(`*Type*`)` Encoding of the document (if any).

`content_length(`*Length*`)` *Length* is the size of the document, in bytes.

`authenticate(`*Challenges*`)` Request for authentication.

- `html2terms(`*Chars*`,` *Terms*`)` We have already explained how this predicate transforms HTML terms to HTML format. Used on the other way it can parse HTML code, for example retrieved by `fetch_url`. The resulting list of HTML terms *Terms* is normalized: it contains only `comment/1`, `declare/1`, `env/3` and `$/2` structures.

For example, a simple fetch of a document can be done as follows:

```
url_info('http://www.foo.com',UI), fetch_url(UI,[],R), member(content(C),R).
```

Note that if an error occurs (the document does not exist or has moved, for example) this will simply fail. The following call retrieves a document if it has been modified since October 2, 1996:

```
fetch_url(http('www.foo.com',80,"/doc.html"),
    [if_modified_since('Wednesday',2,'October',1996,'00:00:00')],R).
```

This last one retrieves the header of a document (with a timeout of 10 seconds) to get its last modified date:

```
fetch_url(http('www.foo.com',80,"/last_news.html"),[head,timeout(10)],R),
member(last_modified(Date),R).
```

The following is a simple application illustrating the use of `fetch_url` and `html2terms`. The example defines `check_links(`*URL*`,`*BadLinks*`)`. The predicate fetches the HTML document pointed to by *URL* and scours it to check for links which produce errors when followed. The list *BadLinks* contains all the bad links found, stored as compound terms of the form: `badlink(Link,Error)` where `Link` is the problematic link and `Error` is the error explanation given by the server.

```
check_links(URL,BadLinks) :-
        url_info(URL,URLInfo),
        fetch_url(URLInfo,[],Response),
        member(content_type(text,html,_),Response),
        member(content(Content),Response),
        html2terms(Content,Terms),
        check_source_links(Terms,URLInfo,[],BadLinks).

check_source_links([],_,BL,BL).
check_source_links([E|Es],BaseURL,BL0,BL) :-
        check_source_links1(E,BaseURL,BL0,BL1),
        check_source_links(Es,BaseURL,BL1,BL).

check_source_links1(env(a,AnchorAtts,_),BaseURL,BL0,BL) :-
        member((href=URL),AnchorAtts), !,
        check_link(URL,BaseURL,BL0,BL).
check_source_links1(env(_Name,_Atts,Env_html),BaseURL,BL0,BL) :- !,
        check_source_links(Env_html,BaseURL,BL0,BL).
check_source_links1(_,_,BL,BL).

check_link(URL,BaseURL,BL0,BL) :-
        url_info_relative(URL,BaseURL,URLInfo), !,
        fetch_url_status(URLInfo,Status,Phrase),
        ( Status \== success ->
          name(P,Phrase),
          name(U,URL),
          BL = [badlink(U,P)|BL0]
        ; BL = BL0
        ).
check_link(_,_,BL,BL).

fetch_url_status(URL,Status,Phrase) :-
        fetch_url(URL,[head,timeout(20)],Response), !,
        member(status(Status,_,Phrase),Response).
fetch_url_status(_,timeout,timeout).
```

9 Providing Code Through the WWW

A facility which can be easily built on top of the primitives presented so far is that of "remote WWW modules," i.e., program modules which reside on the net at a particular HTTP address in the same way that normal program modules reside in a particular location in the local file system. This allows for example always fetching the most recent version of a given library (e.g., PiLLoW) when a program is compiled. The CIAO library provides an extended `use_module` declaration which is identical to the one used in standard systems (e.g., SICStus) but allows using `http` and `ftp` addresses when referring to files. For example, the form handler of Section 6.1, if rewritten as

```
#!/usr/local/bin/lpshell

:- use_module('http://www.clip.dia.fi.upm.es/lib/pillow.pl').

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
...
```

would load the current version of the library each time it is executed. This generalized module declaration is just syntactic sugar, using `expand_term`, for a document fetch, using `fetch_url`, followed by a standard

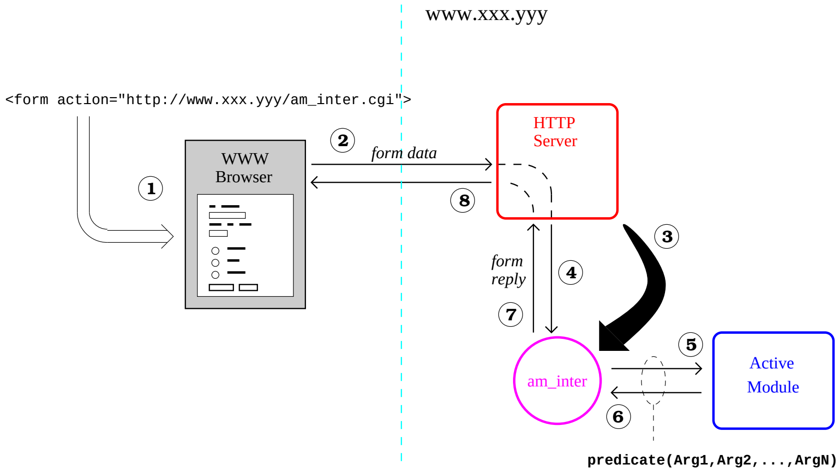

Figure 3: The Forms Interface using Active Modules

`use_module` declaration. It is obviously interesting to combine this facility with caching strategies. An interesting (and straightforward to implement) additional feature is to fetch remote byte-code (as generally done by `use_module`), if available, but this is only possible if the two systems use the same byte-code (this can normally be checked easily in the bytecode itself). Also, it may be interesting to combine this type of code downloading with WWW document accesses, so that code is downloaded automatically when a particular document is fetched. This issue is addressed in Section 11. Finally, there are obvious security issues related to downloading code in general, which can be addressed with standard techniques such as security signatures.

10 A High-Level Model of Client-Server Interaction: Active Modules

Despite its power, the cgi-bin interface also has some shortcomings. The most serious is perhaps the fact that the handler is started and expected to terminate for each interaction. This has two disadvantages. First, no state is preserved from one query to the next. However, as mentioned before, this can be fixed by passing the state through the form or also by saving it in a temporary file at the server side. Second, and more importantly, starting and stopping the application may be inefficient. For example, if the idea is to query a large database or a natural language understanding system, it may take a long time to start and stop the system. In order to avoid this we propose an alternative architecture for cgi-bin applications (a similar idea, although not based on the idea of active modules, has been proposed independently by Ken Bowen [3]).

The basic idea is illustrated in Figure 3. The operation is identical to that of standard form handlers, as illustrated in Figure 2, up to step 3. In this step, the handler started is not the application itself, but rather an interface to the actual application, which is running continuously and thus contains state. Thus, only the interface is started and stopped with every transaction. The interface simply passes the form input received from the server (4) to the running application (5) and then forwards the output from the application (6) to the server before terminating, while the application itself continues running. Both the interface and the application can be written in LP/CLP, using the predicates presented. The interface can be a simple script, while the application itself will be typically compiled.

An interesting issue is that of communication between interface and application. This can of course be done through sockets. However, as a cleaner and much simpler alternative, the concept of active modules [4] can be used to advantage in this application. An active module (or an active object, if modularity is implemented via objects) is an ordinary module to which computational resources are attached (for example, a process on a UNIX machine), and which resides at a given (socket) address on the network.[4] Compiling an active module produces an executable which, when running, acts as a server for a number of relations, which are the predicates exported by the module. The relations exported by the active module can be

[4] It is also possible to provide active modules via a WWW address. However, we find it more straightforward to simply use socket addresses. In any case, this is generally hidden inside the access method and can be thus made transparent to the user.

accessed by any program on the network by simply "loading" the module and thus importing such "remote relations." The idea is that the process of loading an active module does not involve transferring any code, but rather setting up things so that calls in the local module are executed as remote procedure calls to the active module, possible over the network. Except for saving it in a special way, an active module is identical from the programmer point of view to an ordinary module. Also, a program using an active module imports it and uses it in the same way as any other module, except that it uses "`use_active_module`" rather than "`use_module`" (see below). Also, an active module has an address (network address) which must be known in order to use it. The address can be announced by the active module when it is started via a file or a name server (which would be itself another active module with a fixed address).

We now present the constructs used by active modules. Note that for concreteness and compatibility in the description of modules we mainly follow the same scheme as SICStus Prolog.

- `:- use_active_module(`*Module*`,`*Predicates*`)` A declaration used to import the predicates in the list *Predicates* from the (already) active module *Module*. From this point on, the code should be written as if a standard `use_module/2` declaration had been used. The declaration needs the following hook predicate to be defined.
- `module_address(`*Module*`,`*Address*`)` This predicate must give the address of each active module imported in the code.
- `save_active_module(`*Name*`,`*Address*`,`*Hook*`)` Saves the current code as an active module, into the executable file *Name*. When the file is executed (for example, at the operating system level by "*Name* `&`"), *Address* is unified with the address of the module, and *Hook* is called in order to export this address as required.

Note that this scheme is very flexible. For example, the predicate `module_address/2` itself could be imported, thus allowing a configurable standard way of locating active modules. One could, for example, use a directory accessible by all the involved machines to store the addresses of the active modules in them, and this predicate would examine this directory to find the required data. A more elegant solution would be to implement a name server, that is, an active module with a known address that records the addresses of active modules and supplies this data to the modules that actively import it.

From the implementation point of view, active modules are essentially daemons: Prolog executables which are started as independent processes at the operating system level. In the CIAO system library, communication with active modules is implemented using sockets (thus, the address of an active module is a UNIX socket in a machine). Requests to execute goals in the module are sent through the socket by remote programs. When such a request arrives, the process running the active module takes it and executes it, returning through the socket the computed results. These results are then taken by the remote processes.

Thus, when the compiler finds a `use_active_module` declaration, it defines the imported predicates as remote calls to the active module. For example, if the predicate *P* is imported from the active module *M*, the predicate would be defined as

```
P :- module_address(M,A), remote_call(A,P)
```

The predicate `save_active_module/3` saves the current code like `save/1`, but when the execution is started a socket is created whose address is the second argument of the predicate, and the expression in the third argument is executed. Then, the execution goes into a loop of reading execution requests from the socket, executing them, and returning the solutions back through the socket.

Compiling the following program creates an executable `phone_db` which, when started as a process (for example, by typing "`phone_db &`" at a UNIX shell prompt) saves its address (i.e., that of its socket) in file `phone_db.addr` and waits for queries from any module which "imports" this module (it also provides a predicate to dynamically add information to the database):

```
:- module(phone_db,[response/2,add_phone/2]).

response(Name, Response) :-
    form_empty_value(Name) ->
       Response = 'You have to provide a name.'
  ; phone(Name, Phone) ->
       Response = ['Telephone number of ',b(Name),': ',Phone]
```

```
  ; Response = ['No telephone number available for ',b(Name),'.'].

add_phone(Name, Phone) :-
    assert(phone(Name, Phone)).

:- dynamic phone/2.
phone(daniel, '336-7448').
phone(manuel, '336-7435').
phone(sacha,  '543-5316').

:- save_active_module(phone_db,Address,
                      (tell('phone_db.addr'), write(Address),told)).
```

The following simple script can be used as a cgi-bin executable which will be the active module interface for the previous active module. When started, it will process the form input, issue a call to `response/2` (which will be automatically handled by the `phone_db` active module), and produce a new form before terminating. It will locate the address of the `phone_db` active module via the `module_address/2` predicate it defines.

```
#!/usr/local/bin/lpshell

:- use_active_module(phone_db,[response/2]).
:- use_module('/usr/local/src/pillow/pillow.pl').

main(_) :-
    get_form_input(Input),
    get_form_value(Input,person_name,Name),
    response(Name,Response),
    output_html([
        cgi_reply,
        start,
        title('Telephone database'),
        image('phone.gif'),
        heading(2,'Telephone database'),
        --,
        Response,
        $,
        start_form,
        'Click here, enter name of clip member, and press Return:',
        \\,
        input(text,[name=person_name,size=20]),
        end_form,
        end]).

module_address(phone_db,Address) :-
   see('phone_db.addr'),
   read(Address),
   seen.
```

There are many enhancements to this simple schema which, for brevity, are only sketched here. One is to add concurrency to the active module (or whatever means of handling the client-server interaction is being used), in order to handle queries from different clients concurrently. This is easy to do in systems that support concurrency natively, such as &-Prolog/CIAO, BinProlog/μ^2-Prolog, AKL, Oz, and KL1. We feel that &-Prolog/CIAO can offer advantages in this area because it offers compatibility with Prolog and CLP systems while at the same time efficiently supporting concurrent execution of clause goals via local or distributed threads. Such goals can communicate at different levels of abstraction: sockets/ports, blackboard, or shared variables. BinProlog/μ^2-Prolog also supports threads, although the communication mechanisms

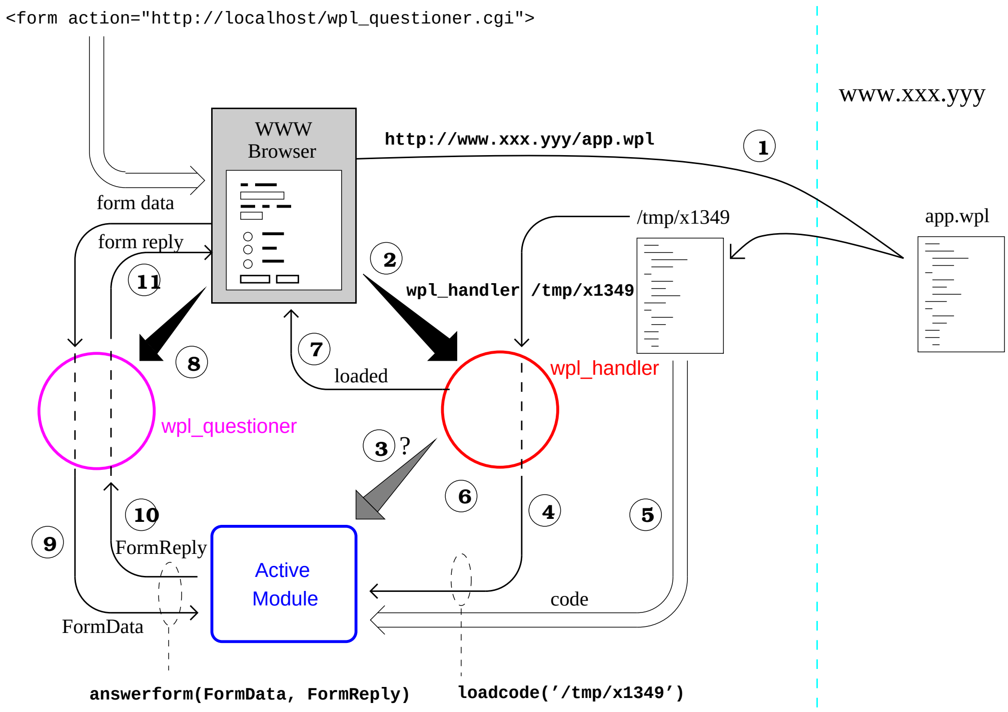

Figure 4: Automatic code downloading architecture

are somewhat different. Finally, as shown in [28], it is also possible to exploit the concurrency present in or-parallel Prolog systems such as Aurora for implementing a multiprocessing server.

It is also interesting to set up things so that a single active module can handle different forms. This can be done even dynamically (i.e., the capabilities of the active module are augmented on the fly, being able to handle a new form), by designating a directory in which code to be loaded by the active module would be put, the active module consulting the directory periodically to increase its functionalities. Finally, another important issue that has not been addressed is that of providing security, i.e., ensuring that only allowed clients connect to the active module. As in the case of remote code downloading, standard forms of authentication based on codes can be used.

An implementation of active modules as described is included in the CIAO library which provides the concurrent and distributed execution facilities mentioned above [4]. As the PiLLoW library, this library only uses standard features of LP/CLP systems, although it does require attributed variables [19] if shared-variable communication is to be used [16]. However, this is not necessary for implementing active modules.

11 Automatic Code Downloading and Local Execution

In this section we describe an architecture which, using only the facilities we have presented in previous sections, allows the downloading and local execution of Prolog (or other LP/CLP) code by accessing a WWW address, without requiring a special browser. This is a complementary approach to giving WWW access to an active module in the sense that it provides code which will be executed in the client machine. More concretely, the functionality that we desire is that by simply clicking on a WWW pointer, and transparently for the user, remote Prolog code is automatically downloaded in such a way that it can be queried via forms and all the processing is done locally.

To allow this, the HTTP server on the server machine is configured to give a specific `mime.type` (for example `application/x-prolog`) to the files which will hold WWW-downloadable Prolog code (for example those with a special suffix, like `.wpl`). On the other side, the browser is configured to start the `wpl_handler` helper application when receiving data of type `application/x-prolog`. This `wpl_handler` application is the interface to a Prolog engine which will execute the WWW downloaded code, acting as an active module. We now sketch the procedure (see figure 4):

1. The form that will be used to query the downloaded code (and which we assume already loaded on

the browser) contains a link which points to a WWW-aware Prolog code file. Clicking on this link produces the download as explained below. Note that for browsers that can handle `multipart/mixed` mime types (such as most modern browsers), the form and the code file could alternatively be combined in the same document. However, for brevity, we will only describe the case when they are separate. The handler for the form is specified as the local cgi-bin executable `wpl_questioner.cgi`.

2. As the server of the file tells the browser that this page is of type `application/x-prolog`, the browser starts a `wpl_handler` and passes the file to it (in this example by saving the file in a temporal directory and passing its name).
3. The `wpl_handler` process checks whether a Prolog engine is currently running for this browser and, if necessary, starts one. This Prolog engine is configured as an active module.
4. Then, through a call to a predicate of the active module "`loadcode(`*File*`)`" the handler asks the active module to read the code.
5. The active module reads the code and compiles it.
6. `wpl_handler` waits for the active module to complete the compilation and writes a "done" message to the browser.
7. The browser receives the "done" message.
8. Now, when the "submit" button in the form is pressed, and following the standard procedure for forms, the browser starts a `wpl_questioner` process, sending it the form data.
9. The `wpl_questioner` process gets this form data, translates it to a dictionary *FormData* and passes it to the active module through a call to its exported predicate `answerform(`*FormData*`,`*FormReply*`)`.
10. The active module processes this request, and returns in *FormReply* a WWW page (as a term) which contains the answer to it (and possibly a new form).
11. The `wpl_questioner` process translates *FormReply* to raw HTML and gives it back to the browser, dying afterwards. Subsequent queries to the active module can be accomplished either by going back to the previous page (using the "back" button present in many browsers) or, if the answer page contains a new query form, by using it. In any case, the procedure continues at 8.

The net effect of the approach is that by simply clicking on a WWW pointer, remote Prolog code is automatically downloaded to a local Prolog engine. Queries posed via the form are answered locally by the Prolog engine.

There are obvious security issues that need to be taken care of in this architecture. Again, standard authentication techniques can be used. However, since source code is being passed around, it is comparatively easy to verify that no dangerous predicates (for example, perhaps those that can access files) are executed. Note again that it is also possible to download bytecode, since this is supported by most current LP/CLP systems, using a similar approach.

12 Related Work

Previous general purpose work on WWW programming using computational logic systems includes, to the best of our knowledge, the publicly available `html.pl` library [5] and manual, and the LogicWeb system [23] (the P*i*LL*o*Wwas described previously in [7]). The `html.pl` library was built by D. Cabeza and M. Hermenegildo, using input from L. Naish's forms code for NU-Prolog and M. Hermenegildo and F. Bueno's experiments building a WWW interface to the CHAT-80 [31] program. It was released as a publicly available WWW library for LP/CLP systems and announced, among other places, in the Internet `comp.lang.prolog` newsgroup [6]. The library has since been ported to a large number of systems and adapted by several Prolog vendors, as well as used by different programmers in various institutions. In particular, Ken Bowen has ported the library to ALS Prolog and extended it to provide group processing of forms and an alternative to our use of active modules [3]. The present work is essentially a significant extension of the `html.pl` library.

The main other previous body of work related to general-purpose interfacing of logic programming and the WWW that we have knowledge of is the LogicWeb [23] system, by S.W. Loke and A. Davison. The aim of LogicWeb is to use logic programming to extend the concept of WWW pages, incorporating in them programmable behaviour and state. In this, it shares goals with Java. It also offers rich primitives for accessing code in remote pages and module structuring. The aims of LogicWeb are different from those of `html.pl`/P*i*LL*o*W. LogicWeb is presented as a system itself, and its implementation is done through a tight integration with the Mosaic browser, making use of special features of this browser. In contrast, `html.pl`/P*i*LL*o*W is a general purpose library, meant to be used by a general computational logic systems and is browser-independent. `html.pl`/P*i*LL*o*W offers a wide range of functionalities, such as syntax conversion between HTML and logic terms, access predicates for WWW pages, predicates for handling forms, etc., which are generally at a somewhat lower level of abstraction than those of LogicWeb. We believe that using P*i*LL*o*W and the ideas sketched in this paper it is possible to add the quite interesting functionality offered by LogicWeb to standard LP and CLP systems. We have shown some examples including access to passive remote code (modules with an `ftp` or `http` address) from programs and automatic remote code access and querying using standard browsers and forms. In addition, we have discussed active remote code, where the functionality, rather than the code itself, is exported.

More recently, a larger body of work on the topic has been presented at the workshop held on the topic of Logic Programming and the Internet at the 1996 Joint International Conference and Symposium on Logic Programming (where also a previous version of this paper was presented). The work presented in [27] is based on LogicWeb, and aims to provide distributed lightweight databases on the WWW. As with the basic LogicWeb system, we believe that the P*i*LL*o*W library can be used to implement in other systems the interesting ideas proposed therein. As briefly mentioned before, the work in [28] proposes an architecture similar to that of our active modules in order to handle form requests. In this solution the handling multiple requests is performed by using or-parallelism. While we feel that and-parallelism is more natural for modeling concurrency, the ideas proposed are quite interesting. The ECLiPSe HTTP-library [25], aimed at implementing INTERNET agents, offers functionality that is in part similar to that of the CIAO `html.pl`/P*i*LL*o*W libraries, including facilities that are similar to our active modules. The approach is different, however, in several respects. The ECLiPSe library implements special HTTP servers and clients. In contrast, P*i*LL*o*W uses standard HTTP servers and interfaces. Using special purpose servers may be interesting because the approach possibly allows greater functionality. On the other hand this approach in general requires either the substitution of the standard server on a given machine or setting the special server at a different socket address from the standard one. The ECLiPSe library also contains functionality that is related to our active modules, although the interface provided is at a lower level. Finally, other papers describing very interesting WWW applications are being presented, which underline the suitability of computational logic systems for the task. We believe that the CIAO P*i*LL*o*W library can contribute to making it even easier to develop such applications in the future.

Additional work on the topic of Logic Programming and the Internet can be found in the proceedings of the workshop sponsored by the Compulog-Net research network. The reader is referred to the tutorials and papers presented in these two workshops for more information on a number of applications, other libraries, and topics such as interfacing and compilation from computational logic systems to Java. An example of a Prolog system interfaced with Java is BinProlog (see `http://clement.info.umoncton.ca/BinProlog`). Experimental Prolog to Java compilers have been built both in academia (see for example jProlog at `http://www.cs.kuleuven.ac.be/ bmd/PrologInJava/`) and Commercially (see for example the IF Prolog tools `http://www.ifcomputer.com`). This approach is quite attractive although the results cannot compete in performance with conventional Prolog compilers (it is open for research whether improvements in Java performance or improved Prolog-to-Java compilation technology can bridge the gap). Other commercial work on the topic of interfacing Prolog and the WWW (in addition to that done on the ALS system mentioned above) include the Amzi! Prolog WebLS System (`http://www.amzi.com/share.htm`) and the LPA PrologWeb System (`http://www.lpa.co.uk`).

A page with pointers to the proceedings of the previously mentioned workshops, as well as other information (including technical reports and tutorial) regarding the topic of Logic Programming, Constraint Programming, and the Internet is maintained at `http://www.clip.dia.fi.upm.es/lpnet/index.html`.

13 Conclusions and Future Work

We have discussed from a practical point of view a number of issues involved in writing Internet and WWW applications using LP/CLP systems. In doing so, we have described P*i*LL*o*W, an Internet/WWW programming library for LP/CLP systems. P*i*LL*o*W provides facilities for generating HTML structured documents, producing HTML forms, writing form handlers, accessing and parsing WWW documents, and accessing code posted at HTTP addresses. We have also described the architecture of some application classes, including automatic code downloading, using a high-level model of client-server interaction, *active modules*. Finally we have also described an architecture for automatic LP/CLP code downloading for local execution, using generic browsers. We believe that the CIAO P*i*LL*o*W library can ease substantially the process of developing WWW applications using computational logic systems.

We are currently working on extended versions of the library which for example may make extensive use of concurrency internally (on those LP/CLP systems that support it) to overlap network requests and include support for (active) VRML (some quite interesting work in this area, in the motivating context of MOOs, is presented in [30]).

In addition to being part of the &-Prolog/CIAO system, the P*i*LL*o*W library is provided as a fully public domain standard library for SICStus Prolog and other Prolog and CLP systems, supporting most of its functionality. Please contact the authors or consult our WWW site `http://www.clip.dia.fi.upm.es` and the P*i*LL*o*W page (`http://www.clip.dia.fi.upm.es/miscdocs/pillow/pillow.html`) for details.

Acknowledgments

The authors would like to thank Mats Carlsson, Tony Beaumont, Ken Bowen, Michael Codish, Markus Fromherz, Paul Tarau, Andrew Davison, and Koen De Bosschere for useful feedback on previous versions of the presented libraries. The first versions of the CIAO system and the `html.pl` library were developed under partial support from the ACCLAIM ESPRIT project.

References

[1] T. Berners-Lee, R. Cailliau, A. Luotonen, H.F. Nielsen, and A. Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[2] K. De Bosschere. Multi–Prolog, Another Approach for Parallelizing Prolog. In *Proceedings of Parallel Computing*, pages 443–448. Elsevier, North Holland, 1989.

[3] K. Bowen. Personal communication, March 1996. Available from `http://www.als.com/als/html_pl.html`.

[4] D. Cabeza and M. Hermenegildo. Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the 1995 COMPULOG-NET Workshop on Parallelism and Implementation Technologies*, Utrecht, NL, September 1995. U. Utrecht / T.U. Madrid. Available from `http://www.clip.dia.fi.upm.es/`.

[5] D. Cabeza and M. Hermenegildo. *html.pl: An HTML Package for (C)LP systems.* Spain, March 1996. Available from `http://www.clip.dia.fi.upm.es/miscdocs/`.

[6] D. Cabeza and M. Hermenegildo. LP/CLP HTML and WWW interface publicly available, February 1996. Posting in `comp.lang.prolog`. Available from `http://www.clip.dia.fi.upm.es/`.

[7] D. Cabeza, M. Hermenegildo, and S. Varma. The P*i*LL*o*W/CIAO Library for INTERNET/WWW Programming using Computational Logic Systems. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet`.

[8] M. Carlsson. *Sicstus Prolog User's Manual.* Po Box 1263, S-16313 Spanga, Sweden, February 1988.

[9] T. Chikayama, T. Fujise, and D. Sekita. A Portable and Efficient Implementation of KL1. In Evan Tick, editor, *Proc. of the 1994 ICOT/NSF Workshop on Parallel and Concurrent Programming.* U. of Oregon, March 1994.

[10] A. Colmerauer. Les gramaire de metamorphose. Technical report, Univ. D'aix-Marseille, Groupe De Ia, 1975.

[11] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[12] European Computer Research Center. *Eclipse User's Guide*, 1993.

[13] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[14] M. Hermenegildo. Writing “Shell Scripts” in SICStus Prolog, April 1996. Posting in `comp.lang.prolog`. Available from `http://www.clip.dia.fi.upm.es/`.

[15] M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Proceedings of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, Portland, Oregon, USA, December 1995. Available from `http://www.clip.dia.fi.upm.es/`.

[16] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *Proc. of the Twelfth International Conference on Logic Programming*, pages 631–645. MIT Press, June 1995.

[17] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[18] M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, LNCS 874, pages 123–133. Springer-Verlag, May 1994.

[19] S. Le Houitouze. A New Data Structure for Implementing Extensions to Prolog. In P. Deransart and J. Małuszyński, editors, *Proceedings of Programming Language Implementation and Logic Programming*, number 456 in Lecture Notes in Computer Science, pages 136–150. Springer, August 1990.

[20] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *ACM Symp. Principles of Programming Languages*, pages 111–119. ACM, 1987.

[21] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[22] R. A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[23] S.W. Loke and A. Davison. Logic programming with the World-Wide Web. In *7th. ACM Conference on Hypertext*, pages 235–245. ACM Press, March 1996. Available from `http://www.cs.unc.edu/~barman/HT96/P14/lpwww.html`.

[24] H. Simonis M. Dincbas and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1 & 2):72–93, 1990.

[25] L. Leth Ph. Bonnet, S. Bressan and B. Thomsen. Towards ECLiPSe Agents on the INTERNET. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP”96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet/lpnet2.html`.

[26] G. Smolka. The Definition of Kernel Oz. DFKI Oz documentation series, German Research Center for Artificial Intelligence (DFKI), November 1994.

[27] A. Davison S.W. Loke and L. Sterling. Lightweight Deductive Databases on the World-Wide Web. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP”96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet/lpnet10.html`.

[28] Péter Szeredi, Katalin Molnár, and Rob Scott. Serving Multiple HTML Clients from a Prolog Application. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP”96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet/lpnet9.html`.

[29] P. Tarau. Binprolog 5.00, April 1996. Posting in `comp.lang.prolog`. Available from `http://clement.info.umoncton.ca/~tarau`.

[30] Paul Tarau and Koen De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996. Available from `http://clement.info.umoncton.ca/~lpnet/lpnet11.html`.

[31] D.H.D. Warren and F. C. N. Pereira. An Efficient, Easily Adaptable System For Interpreting Natural Language Queries. *American Journal of Computational Linguistics*, 8(3-4):110–122, 1982.