Implementing Prolog-Run WWW Sites

Technical Report 2000.04

June, 2000

Department of Information Technology, University of Zurich

Winterthurerstr. 190 – CH 8057 Zurich, Switzerland

Wamberto Vasconcelos* `wvasconcelos@acm.org`
Rolf Schwitter `schwitt@ifi.unizh.ch`
Diego Mollá `molla@ifi.unizh.ch`
João Cavalcanti† `joaoc@dai.ed.ac.uk`

Abstract

We describe a modular and customisable architecture for a WWW server run by Prolog programs and show how each of its components can be implemented. Our proposal employs standard Prolog-CGI technology but to improve efficiency we also use client-server modules to perform the actual services of the WWW site.

1 Introduction

The purpose of this document is to describe a modular and customisable architecture for a WWW server which employs Prolog programs to handle requests from HTML forms. It is our intention to provide as much detail as necessary for the appropriate adaptation and customisation of our architecture to other similar contexts.

Our proposed architecture consists of the following components:

- an HTML form – a web page is offered as an interface to remote users who want to pose requests to the WWW server.
- a Prolog-CGI interface – the request posed by users to the WWW site via the HTML form is handled by a CGI (Common Gateway Interface) application. Our CGI application is a simple Perl program that enables the requests to be processed by a Prolog program.
- a set of client-server modules – in order to process our requests, a Prolog program is initially run to parse and tokenise the actual input and only then the requests can be handled. Since the processing of requests can be quite complex, we suggest splitting these two independent stages (parsing/tokenising and handling the request) into two programs. However, we notice that the program which processes requests can become very large and that loading large programs can take considerable time and degrade the server performance due to simultaneous executions triggered by different requests. We describe how we can use client-server modules to circumvent these problems.

A graphic representation of our architecture is shown in Figure 1 below. Besides the components of our architecture and

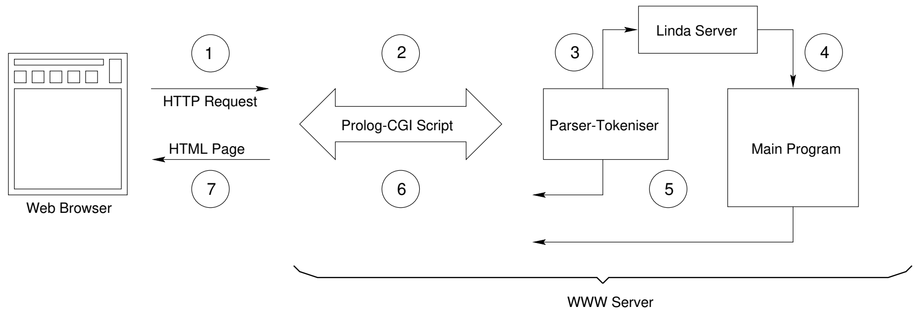

Figure 1: An Architecture for Prolog-Run WWW Servers

their relationships, the diagram also conveys a chronology of the events (numbered circles) involved in an interaction between a user (client) and the main program (server), in a circular fashion. Initially (step 1), a user fills out an

*On a Post-Doctoral leave of absence from Departamento de Estatística e Computação, Universidade Estadual do Ceará, Ceará, Brazil, sponsored by the Brazilian Research Council CNPq, grant no. 201340/91-7.

†Artificial Intelligence, University of Edinburgh, 80 South Bridge, EH1 1HN Edinburgh, Scotland, UK. On leave of absence from Departamento de Ciência da Computação, Universidade do Amazonas, Amazonas, Brazil, sponsored by the Brazilian Research Council CAPES, grant no. 1991/97-3.

HTML form and sends a HTTP request to the WWW Server via a browser. A CGI (Common Gateway Interface) script is activated (step 2) and the request is passed on to a Prolog program to parse and tokenise the request ("Parser-Tokeniser" box). After this, the Parser-Tokeniser (step 3) communicates with a Linda Server Prolog process which, on its turn (step 4) communicates with the main Prolog program ("Main Program" box) that actually handles the request[1]. Either the parser-tokeniser or the main program (or both) can generate output (step 5) which is "piped" to the CGI script. The main Prolog program handles the request and upon its completion the CGI script returns (step 6) whatever output is produced. The CGI script then directs any output to the user's browser who then gets the response to the request (step 7).

In the next sections we describe each of the items above, giving their details and explaining how they can be adapted for different contexts. Throughout this paper we make intensive use of material taken from [5, 7]. However, we expand and improve on these references by recording the experiences and difficulties we had implementing our complete WWW server as well as how problems were solved. Although our discussion here deals specifically with SICStus Prolog [10] running on a Unix platform, we believe readers will find enough details here so as to enable them to adapt our techniques to their pet version of Prolog interpreter or even to another language.

2 The HTML Form

HTML forms are interfaces between users and WWW sites. They come in many different formats ranging from simple click-on buttons to scroll-down menus and areas to type in actual text, etc. on their own and combined together. It is not within the scope of this document to provide readers with an extensive compilation of HTML forms and we suggest [3, 6] for that purpose.

We shall employ a simple form comprising an area and a button as shown in the right-hand side of Figure 2, with its HTML source code on the left-hand side. The user is to type in a piece of text as the "This is a simple input text." sentence and click on the "submit" button. By doing so, the user requests a service from the web site specified by the

```
<html>
<title>HTMLForm</title>
<p>Please type your sentence below and
   click on the ‘‘submit’’ button.
<p><form
     action="http://www.yoursite.com/cgi-bin/script.cgi"
     method="get">
<textarea name="query" rows = "10"
          cols="40" wrap="physical">
</textarea>
<p><input type="submit"
          name="send"
          value="submit">
</form>
</html>
```

$\Longrightarrow$

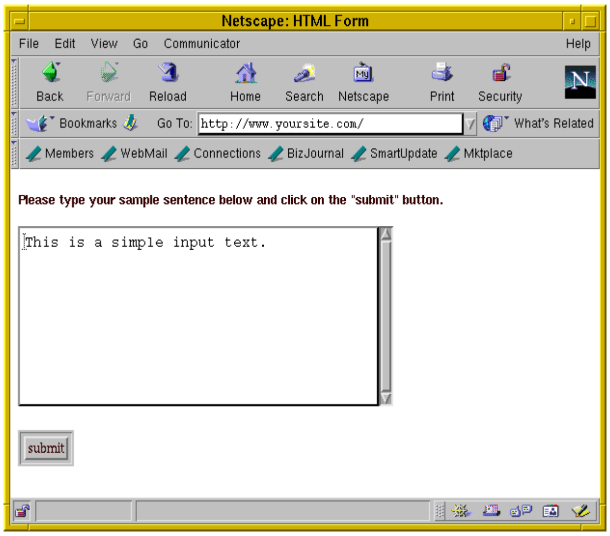

Figure 2: HTML Code (Left) for Form (Right)

form. A CGI script [5, 7, 8], explained in Section 3 below, must be associated with an HTML form. In our running example, we have made use of the `get` method and associated our form with the CGI script `script.cgi` stored in `http://www.yoursite.com/cgi-bin`. When the user clicks on the "submit" button the request query

`http://www.yoursite.com/cgi-bin/script.cgi?query=This+is+a+simple+input+sentence.+&send=submit`

is sent over to the site `www.yoursite.com`. The substring of characters starting with "?" and going until the end comprises the information from the HTML form to be passed on to the `script.cgi` script which is stored in the `cgi-bin` directory of the WWW server machine. However, if the `post` method had been used, the input string would not have appeared in the request query as above, rather it would have appeared in the body of the HTTP request. We explain below some important technical details of scripts.

It should be pointed out that regardless of the method used at least one `INPUT` element of type `SUBMIT` or `IMAGE` is necessary in all forms. There are two methods to submit forms. Depending on which method is used the encoded results of the form is passed to the CGI program in a different way. This issue is explained in Section 3.

3 The CGI Script

A CGI script is a special kind of program started via an HTML form and run on the WWW site's machines. Scripts are sequences of commands like ordinary programs, however their commands consist of instructions at the operating

[1] In Section 4 we justify the need for this set of client-server modules to implement the WWW server. At this point we simply say that it is concerned with efficiency issues.

system level such as handling files (deleting or printing them, for instance) or running other scripts and programs in programming languages like C or, in our case, Prolog. We suggest [8] for a more thorough explanation on the Common Gateway Interface.

Any programming language which can read environment variables and which provides executable code (like C, C++ or Java) can be used as a CGI application. Prolog, being a high-level symbolic language, is an excellent tool for programming WWW applications [1, 2, 11], but some implementations do not allow the creation of executable code. Furthermore, in the standard Prolog definition [9] there is no way to refer directly to environment variables. That is why, if we want to use Prolog as a programming language for WWW applications, we have to provide an executable program (*i.e.* a binary or script file) that can invoke the Prolog interpreter, load a given program and run it. For the sake of convenience, this executable will also deal with the CGI particularities. We have used a CGI script [5, 7, 8] for this purpose[2] and we proceed below to describe this approach.

A first piece of technical information is that a CGI script, in order to handle requests of HTML forms, has to be stored in a special folder, *viz.* the cgi-bin sub-folder of the WWW server machine, otherwise it *just does not work*! In some sites, where security against invaders is a priority issue, very strict policies are enforced concerning access to cgi-bin folders and what can be made available in them. For instance, some sites may demand that the CGI script be thoroughly tested before they are made available to outside users. We recommend that readers interact directly with the computing support and WWW team of their sites to find out more about such restrictions. If it turns out to be a delicate political issue to have a CGI script available via normal means, one can always set up one's own independent WWW server (for instance, using free software like Jigsaw – see `http://www.w3.org/Protocols/` for more information).

Another important technical issue concerns the *method* employed in the HTML form. Forms are used in various contexts as a standardised mechanism for bidirectional data exchange over the web [6]. Methods are used to specify how the form is to be handled by the CGI script and they (method and CGI script) are closely related. We have experimented with two methods, `post` and `get`, and their respective CGI scripts, explained below.

If you use method `get` in your form, specified by the following HTML command

```
<form action="http://www.yoursite.com/cgi-bin/script.cgi" method="get">
```

as part of your HTML web page then your CGI program will receive the encoded form input in the environment variable `QUERY_STRING`. Since the size of environment variables content is limited, you might not be able to use `get` method if the query is too long. Supposing we have access to the cgi-bin folder of our WWW site, we shall store our `script.cgi` file with the contents depicted in Figure 3; the numbering is not part of the actual contents and have been used to make references to specific portions of the script easier. The first line informs the operating system

```
1  #!/bin/perl
2  print "Content-type: text/html\n\n";
3  open(TMP, 'bin/sicstus -r www/cgi-bin/parser_token -a $QUERY_STRING |');
4  while (($in = <TMP>))
5  { $all = $all .  $in;
6  }
7  close(TMP);
8
9  print "$all";
```

Figure 3: Commands of CGI Script `script.cgi` (`get` Method)

that the CGI script makes uses of the scripting language Perl. We shall not go into details as to what precisely the components of the syntax of each line mean; we refer readers to [12, 13] for a more comprehensive study on Perl's form and capabilities. We provide here, though, an overall explanation for each command. The second line prints the preamble to the output which is essential since it will be displayed on a browser as HTML text – if such a preamble is not printed then the output will be displayed in the browser not as hypertext but simply as a string of characters (without the formatting/typesetting, graphics, hyperlinks, etc.). The third line consists of a command to load SICStus (`bin/sicstus`) and restore (flag `-r`) the saved program `parser_token` (stored in `www/cgi-bin`, the cgi-bin folder, otherwise it won't work!) using as an input argument (flag `-a`) the substring from the HTML form starting after the "?", referred in Perl simply as `$QUERY_STRING`. This corresponds to step 2 in our diagram of Figure 1. Any output resulting from running SICStus with restored program `parser_token` should be "piped" (that is, directed) to temporary area `TMP` as indicated by the "`|`" symbol and the `TMP` as the first argument of `open`. The fourth, fifth and sixth lines define a simple loop transferring the contents of `TMP` to variable `$all` and line 9 prints the contents of `$all`, that is, any output of the computations arising after running the command of line 3, onto the browser of the user who submitted the HTML form. This output process corresponds to step 6 of our diagram shown in Figure 1.

The contents of the CGI script shown above should remain unchanged, with the exception of the path to both SICStus and the saved program which should be adapted accordingly. The *full* path to these components should be provided within Perl scripts, even though the installation has default paths. The names of the files for the CGI script and the parser-tokeniser can, of course, be changed.

[2] Another alternative would be to write a C or Java program to deal with CGI and interface this program with an executable Prolog program. Although we know that in principle this approach is feasible we haven't investigated it thoroughly.

If, however, you use method `post` in your HTML form, including the line

```
<form action="http://www.yoursite.com/cgi-bin/script.cgi" method="post">
```

as part of your HTML form then your CGI program will receive the encoded form input on `STDIN`, that is, the standard input. The server will *not* send you an `EOF` (end of file) at the end of the data. Instead you should use the environment variable `CONTENT_LENGTH` to determine how much data you should read from `STDIN`. The CGI script corresponding to the `post` method is presented in Figure 4.

```
1  #!/bin/perl
2  print "Content-type: text/html\n\n";
3  read(STDIN, $buffer, $ENV'CONTENT_LENGTH');
4  open(TMP, "bin/sicstus -r www/cgi-bin/parser_token -a $buffer |");
5  while (($in = <TMP>))
6  { $all = $all .  $in;
7  }
8  close(TMP);
9  print $all;
```

Figure 4: Commands for CGI Script `script.cgi` (`post` Method)

Basically, the differences between the `get` method script and the `post` one are lines 3 and 4. In line 3 a sequence of bytes defined by `CONTENT_LENGTH` is read in from `STDIN` and stored in variable `$buffer`. This variable is passed on to our SICStus Program via command line 4. One should notice that the variable `$buffer` is used in place of the environment variable `QUERY_STRING` of our previous script for the `get` method.

An essential issue concerns the format of the output of our programs. Since the output is to be viewed through a web browser, its format should reflect this, with suitable HTML typesetting commands and tags. It is important that the very first string to be output should be "`Content-type: text/html`" *followed by at least one blank line* which works as a preamble to an HTML document. This has been ensured in our scripts above by their second line. Without this command the output will be shown as a string without any HTML formatting even though the HTML tags are in the appropriate places. Caution should be employed when programming so as to make sure that the first thing to be output is the string above.

3.1 The Parser-Tokeniser

The above CGI script interfaces the HTML form with a Prolog program. The restored Prolog program, the parser-tokeniser, is run with the contents of the HTML form. Each user who submits a request via our HTML form above will trigger an execution of the CGI script on the WWW server machine.

In order to provide a quick and smooth upload of the Prolog program that initially handles the HTML form we *restore* a program whose state has been previously saved. This is quicker than loading the actual program. Besides, when we save a program, we can specify the predicate that is to be invoked whenever the program is restored. More details on saving and restoring SICStus Prolog programs can be found in [10].

We show in Figure 5 a simplified edited version of our parser-tokeniser with its overall structure (again, numbers have been used to make references to specific lines easier). The first line informs SICStus that two of its library modules, `linda/client` and `system`, are to be loaded for the program to be run. As mentioned above, our architecture employs Prolog client-server technology to improve efficiency of the WWW site, hence the need to load `linda/client`. The predicate `main`/0 defined from line 2 to 9 is to be invoked initially when the program is restored (this is specified by one of the arguments of SICStus `save_program`/2 built-in, explained below).

Predicate `main`/0 starts off (line 3) by importing into the program the contents of the CGI script's variable `$QUERY_STRING`, that is, the contents of the HTML form; built-in `prolog_flag`/2 is part of the library `system`, hence the need to load it in line 1. The contents of the HTML form is made available as string of characters assigned to variable `Arg`, all performed by `prolog_flag`/2. Lines 4–8 comprise two nested if-then-else commands which tokenise and parse the input string into a list of keys, handling the potential errors by appropriate messages (predicate `displayMessage`/2 pretty prints messages and prepares the HTML page taking care of all its details). We have used the code provided in [5] for the actual parsing and tokenising (predicates `tokenizeatom`/2 and `parse_cgi`/2). If the input string has successfully been tokenised and parsed, then we proceed to predicate `process`/1 carrying with us a list of keys each of which is of the form `key(`*`KeyName`*`,`*`KeyContent`*`)`. Again, more details can be found in [5].

The predicate `process`/1 handles a specific list of keys as shown in the pattern of its arguments. This predicate simply establishes a communication link between the parser-tokeniser and the main Prolog program that actual handles the HTML forms. In this sense, the parser-tokeniser is like a "receptionist" that receives the input from the CGI script, makes sure this is OK then passes it on to the main program. Lines 11–18 comprise the usual sequence of commands for achieving Linda client-server communication [10]: initially (line 11) the file `server.addr`, previously recorded (see below for more details) is opened and its contents, of the form `Host:Port` is read (line 12) and then the file is closed (line 13). A communication channel (called a "handle") is then opened (line 14) between the program `parser_token` and the Linda server, a message is passed (line 15) to the server via `out`/1 and the control of execution waits for the server response which is received in line 16. The "`Result`" is processed in line 17 (an adequate definition for this

```
1  :- use_module(library('linda/client')), use_module(library(system)).

2  main:-
3    prolog_flag(argv,[Arg]),
4    (tokenizeatom(Arg,TokenList) ->
5      (parse_cgi(TokenList,KeyVals) ->
6        process(KeyVals)
         ;
7        displayMessage(Arg,'Could not be parsed'))
       ;
8      displayMessage(Arg,'Could not be tokenised')),
9    halt.

10 process([key(query, Query),key(send,[submit])]):-
11   see('server.addr'),
12   read(Host:Port),
13   seen,
14   linda_client(Host:Port),
15   out(query(Query)),
16   in(result(Result)),
17   process_result(Result),
18   close_client.

19 my_save:- save_program(parser_token, main).
```

Figure 5: Edited Version of Parser-Tokeniser `parser_token.pl`

predicate should be supplied as part of your Parser-Tokeniser program). Finally (line 18) the communication channel is closed.

For the sake of completion, we have also included a `my_save`/0 predicate which saves the program (informing the predicate that should be initially invoked, that is, `main`/0) in order for it to be restored by the CGI script. Once the program `parser_token.pl` is loaded in SICStus, the command `my_save` should be typed in and a (large-ish) file will be created as `parser_token`. This is the file that should be stored in the cgi-bin folder, because the CGI script will need it.

An altogether different alternative to this approach is to use existing Prolog libraries (such as PiLLoW/CIAO [4]) as ready-to-use Prolog code to handle CGI and long HTML typesetting commands. Such ready-made macros/subroutines can save a great deal of time (programming and debugging) and allow for a cleaner programming style. However, as is the case with all ready-to-use libraries, some time must be invested in learning to correctly use their constructs. Moreover, when more sophisticated or special-purpose programming is being carried out, limitations and/or idiosyncrasies of the library may be found.

4 A WWW Server as Client-Server Modules

Our WWW server architecture is based on client-server modules. Our design decision was based on performance concerns. One could use a single program to deal with both the reception and the actual dealing of the HTML form – in such case it would not be necessary to use any of this client-server technology. Our Prolog program of Figure 5 could be altered in a straightforward fashion: the body of `process`/1 would contain the actions related to the HTML form. However, we envisage a not-so-simple setting in which our assumptions and constraints are:

1. We suppose that our site will need a large Prolog program to handle all its services.

2. We have adopted CGI scripts as a means to interface the network protocol with Prolog.

3. CGI scripts can only do as good as the operating system allows them to: the only way to run a Prolog program is by initially invoking SICStus then loading the program. However, rather than loading the program code and interpreting it, we can do better by *restoring* a *saved state*, thus making the uploading a lot more efficient. We have profitably made use of this trick in our small "receptionist" program `parser_token` above.

4. Notwithstanding, our main program is large enough to make even the saved state/restore trick not much help. Saved states can get very large if the source program is large; loading big files can take considerable amounts of time.

5. Furthermore, for each user that submits an HTML form a copy of `parser_token` is run. If there are too many requests being dealt at the same time (and each request requires a huge file to be loaded) then this may seriously affect the performance of the server. Although in remote applications the overhead of data transfer across narrow bandwidth (slow transfer rate) can overshadow any other delays, the degradation of the server performance is not irrelevant.

With this rationale, we have split our WWW server into two parts: a first simple part that receives the input from the CGI script (checks its correctness) and a second more sophisticated part that handles the form request. The former will have as many copies running as there are requests; the latter will have only one copy *permanently* running on the server, and handling the requests from our simple programs. The simple programs are started, run, interact with the main program of our server, and then disappear; our main program will persist all the way through.

4.1 The WWW Server Main Program

In this section we show and explain the actual contents of our main program. In Figure 6 we can see the Prolog source code for the main program, `main.pl`. The first line tells SICStus that the `linda/client` library is used in the

```
1   :- use_module(library('linda/client')).

2   start_main:-
3     see('server.addr'),
4     read(Host:Port),
5     seen,
6     linda_client(Host:Port),
7     main_loop.

8   main_loop:-
9     repeat,
10      in(query(Query)),
11      services(Query,Answer),
12      out(result(Answer)),
13    fail.
14
15  services(Query,Answer):- ...

16  :- start_main.
```

Figure 6: Edited Version of WWW Main Program `main.pl`

program – our main program is another Linda *client* (and not a server!) process but contrary to the `parser_token` program, it will run permanently. The `start_main/0` predicate, similar to what happens in the `process/1` predicate of `parser_token.pl` program, reads in the `Host:Port` address through which the communication between `parser_token.pl` and `main.pl` will take place. In line 7, we have a call to `main_loop/0`, defined in lines 8–13: that predicate is an endless loop receiving requests (line 10), processing them (line 11) and outputting the result (line 12). One should notice the reverse order in the sequence of `in/1` and `out/1` subgoals in the programs. The predicate `services/2` is the "core" of the server, where the processing of the query actually takes place. Line 16 starts up the server and should be the very last line of the file: this command is necessary when bootstrapping the server, as we shall see below.

"Blockages" or long delays may happen during the main loop (lines 8–13) if too many copies of the CGI script simultaneously try to communicate with `main.pl`. Blockages may happen if a request makes the main program stop and return an error. This can be handled by catching all the exceptions that occur while processing a request. Long delays, on the other hand, may happen if a request takes too long to process: all the other (posterior) requests will have to wait for the long one to finish. Timeout and exception handling can be implemented with SICStus' module "timeout" and SICStus' exception handling mechanisms [10]. In Figure 6 timeout and exception handling routines would have to be part of predicate `services/2` in line 11.

4.2 Bootstrapping the WWW Server

We explain here the two-step procedure for bootstrapping our WWW server, implemented as a set of Linda client-server processes. The two programs `parser_token.pl` and `main.pl` will be run as Linda *clients*. A third *server* process is necessary to establish the communication channels (*i.e.* the "handle") between them. The set comprising these three processes comprises our WWW server (plus, of course, the Prolog-CGI Script – see Figure 1). There is no need to write yet another Prolog program for this purpose. The following bootstrapping command, to be run in the WWW server machine is enough to start up our Linda server process:

```
echo "use_module(library('linda/server')),
      linda((Host:Port)-(tell('server.addr'),
                         write('\''),write(Host),write('\':'),write(Port),write('.'),
                         told))." | sicstus &
```

Although we have prettyprinted it above to improve visualisation this command should be all in one line otherwise it won't work! The command above invokes SICStus and pipes in a query. The query is the actual bootstrapping program to start up the Linda server and to record the host name and port number in the `server.addr` file. The "'" around the host name are *essential* if the site employs machine names containing special characters or dots. Since they will not interfere when the names are strings of simple characters just leave them there. After the above command is executed, a SICStus process will run in the background and the file `server.addr` will be open with the `Host:Port` information stored in it.

After the above command is executed (and only after it, otherwise it won't work!) we bootstrap the server itself. It should run in the background, hence the need to bootstrap it. The following command should be typed for that purpose:

```
nohup echo "[main]." | sicstus &
```

It simply invokes SICStus and loads in program `main.pl` shown in Figure 6. Notice that line 16 will cause the execution of predicate `start_main/0` upon the upload of the file (and hence that start-up command should be at the very end, or any remaining clauses won't be loaded).

5 Final Remarks and Summary

In this document we have defined, explained, and justified an architecture for a WWW Server fully run by Prolog programs. We have tried to provide sufficient implementational details so as to allow the reproduction and adaptation of our experiments. The actual code of the programs has been edited for the sake of saving space, but all the essential parts have been kept.

In order to deal with efficiency issues, we have split our server into two specialised programs, `parser_token.pl` and `main.pl`, the former responsible for the reception and checking of the HTTP request (provided by the HTML form) and the latter concerned with providing the actual services of the WWW site, and connected these by means of client-server modules. By splitting our WWW server into these two programs we are able to circumvent the overhead of uploading huge files to process each request. It may be argued that this division is subjective and that the actual parsing/tokenising could be performed by the server or, conversely, that some simpler services could be handled by the initial program. However, one should try and keep the initial program as small as possible because each time a request is posed via an HTML form a new process will be started in the WWW server machine thus taking up time and computational resources.

A summary of this paper with suggestions on what one should do to adapt our ideas is as follows:

1. A CGI script interfaces the HTML form with a simple Prolog program `parser_token.pl`. The contents of the CGI script should not be changed, with the exception of the paths for SICStus and the cgi-bin directory.

2. A simple Prolog program `parser_token.pl` receives the HTML form, checks its format and issues appropriate messages if contents are not correct. This program makes use of SICStus Linda Client-Server technology. There is no need to change it unless an altogether different architecture is envisaged. Once the program is stable, it should be *saved* (to be later *restored* by the CGI script).

3. The actual WWW server handling the queries is programmed as `main.pl` which communicates with `parser_token.pl` via Linda Client-Server technology. This program should embed all the processing of the site and the predicate `services/2` should be adequately programmed for that purpose.

4. The bootstrapping procedure should be carried out in the sequence above: first the Linda Server process should be triggered off to establish the handle through which `parser_token.pl` and `main.pl` can communicate, then the server should be started off. There's no need to change any of the bootstrapping steps. The SICStus process running the server should never stop – when that happens, the WWW server will be knocked out. This will happen if the machine running the processes crashes, in which case the bootstrapping procedure will have to be performed again (the bootstrapping commands could be included in the batch file used to re-start the machine).

References

[1] H. Boley. Knowledge Bases in the World-Wide Web: a Challenge for Logic Programming. Technical Memo TM-96-02, Deutsches Forschungszentrum für Künstliche Intelligenz, October 1997.

[2] H. Boley. Beziehungen zwischen Logikprogrammierung und XML. In *Workshop Logische Programmierung*, Würzburg, Germany, Jan 2000. published as GMD Report 90.

[3] M. Bryan. *SGML and HTML Explained*. Addison-Wesley, U.S.A., 1997.

[4] D. Cabeza, M. Hermenegildo, and S. Varma. WWW Programming using Computational Logic Systems (and the PiLLoW/CIAO Library). Technical report, Computer Science Department, Technical University of Madrid, 1996.

[5] B. Carpenter. A Prolog CGI Interface. `http://www.colloquial.com/tlg/cgi.html`, 1999.

[6] W3 Committee. HyperText Markup Language Home Page. `http://www.w3.org/MarkUp/`, February 2000.

[7] M. Grobe and H. Naseer. An Instantaneous Introduction to CGI Scripts and HTML Forms. `http:// www.cc.ukans.edu/ acs/docs/other/ forms-intro.shtml`, 1994.

[8] NCSA HTTPd. The Common Gateway Interface. `http://hoohoo.ncsa.uiuc.edu/cgi/`, 1998.

[9] ISO. ISO Prolog Standard. Technical Report ISO/IEC DIS 13211-1:1995(E), International Organisation for Standardisation, Geneva, Switzerland, 1995.

[10] Intelligent Systems Laboratory. SICStus Prolog User's Manual. Swedish Institute of Computer Science, available at `http:// www.sics.se/isl/sicstus2.html#Manuals`, February 2000.

[11] S. W. Loke and A. Davison. Logic Programming with the World-Wide Web. In *Proc. Hypertext'96*, pages 235–245, Washington, DC, 1996. ACM.

[12] R. L. Schwartz, T. Christiansen, and L. Wall. *Learning Perl*. O'Reilly & Associates, U.S.A., 2nd edition, 1997.

[13] L. Wall, T. Christiansen, R. L. Schwartz, and S. Potter. *Programming Perl*. O'Reilly & Associates, U.S.A., 2nd edition, 1996.