Acta Cybernetica **18** (2008) 561–578.

Prosper: Developing Web Applications Strongly Integrated with Prolog

Levente Hunyadi*

Abstract

Separating presentation and application logic, defining presentation in a declarative way and automating recurring tasks are fundamental issues in rapid web application development. Albeit Prolog is widely employed in intelligent systems and knowledge discovery, creating a web interface for Prolog has been a cumbersome task producing poorly maintainable code, which hinders harnessing the power of Prolog in information systems. This paper presents a framework called Prosper that facilitates developing new or extending existing Prolog applications with a presentation front-end. The framework relies on Prolog to the greatest possible extent, supports code re-use, and integrates easily with web servers. As a result, Prosper simplifies the creation of complex, maintainable web applications running either independently or as part of a heterogeneous system without leaving the Prolog domain.

Keywords: Prolog, web application development framework, application integration, XML, heterogeneous systems

1 Introduction

In developing information systems, modelling complex business processes is a challenging task. The model has to cater for many exceptions to a general rule, has to adapt to current business demands and has to be able to cope with large volumes of heterogeneous data sources. Flexibility and quick development are key issues. On the other hand, Prolog can ease development in a variety of ways. By straightforward formalisation of business rules, it yields verifiable code that is still close to the application domain. Extended BNF-style grammars contribute to the flexibility of data transformations between different data pools. In addition, constraint programming and intelligent reasoning based on background knowledge are other fields where compact Prolog programs can be formulated for complex problems. In other words, the expressiveness of Prolog can contribute greatly to the development of information systems.

*Budapest University of Technology and Economics, Department of Automation and Applied Informatics, H-1111 Budapest, Goldmann György tér 3., Hungary. E-mail: hunyadi@aut.bme.hu

Using Prolog in information systems inevitably requires integration with other parts of a larger system. However, the execution model of Prolog and imperative languages differs substantially, making it difficult to embed Prolog code in a program written in C#, Java or C++. Even though libraries [16, 14] are available to support integration to some degree, the resultant code is often very obscure, type safety is not enforced and debugging code is problematic.

One solution to this problem is presentation-level integration, indicated by the proliferation of XML and web services. In this approach, it is not the applications themselves that are integrated but output produced by one application is consumed by the other, allowing the use of completely different programming languages in the process.

An interesting application field of presentation-level integration is component-based development of web portals. In this case, portals are not built as monolithic applications but are composed of small, fairly independent components, called web parts or portlets. Each component generates presentation-level code (e.g. XHTML or XML), which is combined into a whole by a web portal framework. In this scenario, Prolog can be used to generate parts of the portal that exhibit intelligent behaviour while the rest can be developed by means of conventional imperative programming languages.

The idea of generating HTML or XML output in Prolog is not new: several frameworks [5, 14, 10, 11] exist that give excellent support for structured display of data in web pages. However, neither of them promotes a clear definition of presentation (i.e. how data are displayed) that is distinct from application logic (i.e. the main job of the application). As a result, presentation and application (or business) logic are interleaved, which in most cases eventually leads to poor maintenance. In addition, complex presentation logic, such displaying parts of a page based on a condition, or displaying variations of a given content repetitively for each element of a list are tasks that cannot be accomplished in a generic manner. Moreover, it would be desirable that these tasks be purely restricted to authoring XHTML or XML documents, possibly by using special annotation.

The proposed system, named *PROlog Server Pages Extensible aRchitecture* (or *Prosper* in short) [8], aims to combine the advantages of conventional web application development methods (such as separation of presentation and application logic and declarative definition of presentation by means of XML) with the potential in the Prolog language in order to create more intelligent web portals. It supports integrating Prolog applications in existing information systems as well as extending existing Prolog applications with a web interface.

Prosper is implemented mainly in SWI-Prolog and partially in C. SWI-Prolog is compliant to part one of the Prolog ISO standard and has comprehensive support for multi-threading. ISO-compliance caters for portability while multi-threading helps harness the potential in parallel execution. Network communication interfaces have been written in C to ensure maximum performance. The Prosper project (including full source code) is available at SourceForge.net [13].

The rest of the paper is structured as follows. Section 2 gives a brief introduction to methodologies and technologies the proposed framework makes use of.

Section 3 elaborates on design trade-offs, inspects related work and analyses possible approaches to create a framework for developing a web front-end with special attention to the chosen approach. In Section 4, the architecture of the proposed framework is laid out. Communication of remote parties over a network can be broken down into a series of requests and associated replies: Section 5 traces the way a request produces a reply in Prosper by means of an example. Section 6 gives some implementation details and performance metrics while Section 7, with which this paper concludes, summarises contributions and outlines possible ways of extension and future work.

Throughout the paper, knowledge of Prolog with basics on SGML-languages (especially XML [3] and (X)HTML [12]) and some experience in developing web applications with an application development framework (such as ASP.NET [9] and/or Java [2]) is assumed. One should consult the indicated sources for further details on these technologies.

2 Background

Essentially, the web operates in a request-and-reply manner according to the HyperText Transfer Protocol [6]. First, the client formulates a *request* querying a document. The request is received by the server, which looks up the requested document in its file system and returns it to the client in *reply*. In the case of *dynamic content generation*, the request received by the server does not correspond to a file system entry but is forwarded to a possibly external application that outputs the reply based on the *request context* (request parameters, session information, user preferences, etc.). Web application development frameworks are inserted into the chain either in place of the server (e.g. Java web solutions) or between the server and the external application (e.g. the ASP.NET framework), and expose a programmer-friendly view of the web environment to the application developer.

Web frameworks taking the place of the server require a thorough implementation to provide general web service functionality (e.g. include serving static content) with sufficient security. For this end, it is often desirable to use a trusted web server behind which applications are placed rather than using a separate endpoint for each application. In this scenario, frameworks are often connected to servers by means of server APIs (application programming interfaces). Here, the application is loaded as a module of the server and the server forwards requests that match some criteria (e.g. URL pattern or extension) to the application instead of processing them itself. This is called *strong coupling*.

Common Gateway Interface (CGI) describes a protocol that provides *loose coupling*. In order to process matching requests, the server invokes an external application (i.e. one that is not integrated with the server) with the given request context (query parameters, user settings, etc.) and returns the output it produces to the client. Loose coupling separates the operating system processes involved, which therefore minimally affect each other, increasing flexibility. In addition, fatal errors in the application do not endanger the server. Nonetheless, repetitive invocation of

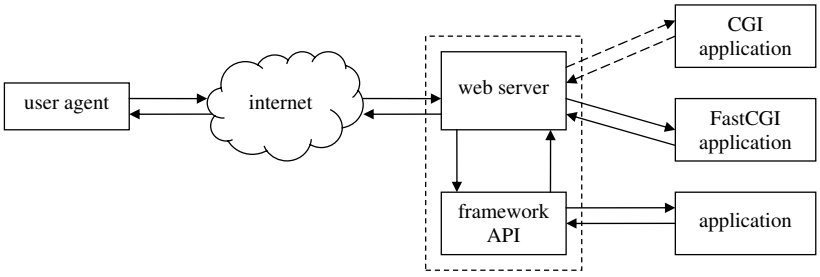

Figure 1: The web service chain. The dashed rectangle indicates the boundary of process space, continuous arrows refer to persistent, while dashed arrows to non-persistent connections.

an external program can take up valuable resources (by successive process initialisations, re-opening database connections, etc.). FastCGI [4] is a persistent version of the CGI protocol that allows applications to remain active after serving a request (thereby maintaining database connections, etc.) yet preserving their independence (i.e. no modification of server internals is required and the application works with web servers of multiple vendors).

The request and response chains and the relationships of the various application types are shown in Figure 1.

Applications that process many simultaneous requests have to be *multi-threaded* so that processing a request does not keep others waiting. Hence, each request is assigned a separate newly initialised thread. However, on high demand this can lead to the so-called *thrashing*, where threads with already assigned jobs lose computing resources to dozens of rapidly launched new threads, eventually leading to no thread performing useful task. Thus, applications often make use of the *worker thread model*. In this model, a constant number of threads execute concurrently. Jobs are assigned threads from a pool, which return to the pool after the job is complete. This allows fast processing of simultaneous requests with the elimination of thread startup costs and stability upon high demand.

Many web development platforms make use of the *model-view paradigm*. In this paradigm, *application logic* (what the program does) and visual *presentation* (how the results are displayed) are strictly separated. Presentation is defined in a declarative manner, often by means of a markup document (such as XML or XHTML), albeit additional code that drives the presentation layer (such as data binding or events) may be required in a *code-behind* file. On the other hand, application logic is written in the native language of the platform. While presentation may reference objects in application logic, the reverse is not true. This allows presentation and application logic to be created (more) independently and caters for easier maintenance of both. While not every web development framework makes it compulsory, the pattern can be considered fairly wide-spread, and is recognised as a key condition to creating complex web applications.

```
env(html, [], [
  env(body, [lang=hu],
    heading(1, title),
    text,
    ref('http://www.aut.bme.hu', hyperlink)
  ])
])
```

```
<html>
  <body lang="hu">
    <h1>title</h1>
    text
    <a href="http://www.aut.bme.hu">hyperlink</a>
  </body>
</html>
```

Figure 2: A PiLLoW Prolog term (above) and the equivalent HTML document it produces (below, ignoring white space).

3 Possible approaches and related work

In order to expose a web front-end, an application has to emit XML or (X)HTML documents based on application logic. This can be accomplished in two different ways:

1. producing and emitting presentation markup code directly using the platform language, or
2. embedding snippets of code written in the platform language within presentation markup code.

Generating web content directly in a Prolog program (1st approach), possibly with the help of general-purpose libraries, is fairly straightforward. The PiLLoW library [5], available in many Prolog implementations, is a notable representative of this approach. As exemplified by the library, the close relationship of Prolog terms and the hierarchical structure of HTML easily lends itself to composing the web page in the form of terms (Figure 2), which are then transformed to and output as plain text on demand. By means of uninstantiated variables in the term representation, simple templates can be created.

Nevertheless, a Prolog term representation is inherently not visual and integrates poorly into existing web authoring tools (such as web page designers). Moreover, the approach does not promote clear separation of application logic and presentation, so that programmers are tempted to violate the model-view paradigm, which eventually leads to more difficult maintenance. Also, a stand-alone Prolog server replying to requests on a dedicated port is often assumed, which is hard to

```
<?, member(number=N, Get),
    forall( ( between(1, N, X), factorial(X, Y) ),
?>
            <li>The factorial of <?= X ?> is <?= Y ?>.</li>
<?  ), ?>
```

Figure 3: An excerpt from a dynamically generated HTML server page composed with embedded Prolog escape sequences. The snippet lists all factorials from 1 to *N*. *N* is specified as a query string parameter.

incorporate into a complex environment with an existing web server. However, a library such as PiLLoW can relieve the programmer from the majority of recurring tasks and can thus contribute greatly to web application development, especially in simple scenarios. Commonly aided tasks include parsing HTTP GET and POST parameters, generating forms, HTTP cookies and session maintenance.

Embedding pieces of Prolog in the presentation layer (2nd approach) is another natural approach, which can be thought of as the "inside out" version of the previous one, motivated by various successful server-side technologies such as PHP [1]. Here, web pages are composed as (X)HTML rather than as Prolog terms, and Prolog calls are inserted in the text by means of special escape sequences. The helper library parses the page into a predicate consisting of a series of *write/1* statements and the equivalents of the embedded Prolog calls. Many projects that take this approach exist in the Prolog domain, [10] and [11] are two such examples.

Albeit simple, this approach is generally insufficient for larger projects as it is weakly structured. Apparently, even repetitively displaying a variation of a block of text as in Figure 3 produces code that is difficult to comprehend. More complex nesting is even harder to implement merely by means of skipping in and out of escaped blocks. Clearly, escape sequences lead to interleaved application logic and presentation, and are hence extremely difficult to maintain or extend.

Another variant of the second approach is composing web pages in an external framework, such as JSP or ASP.NET, and embedding foreign language calls to Prolog. PrologBeans for Java and PrologBeans.NET for the .NET platform [14], both available as SICStus extensions, are representatives of this variant. Here, all web-related issues are handled by an external framework, which provides optimised solutions to general patterns in web authoring and offers rapid application development. In order to call Prolog predicates, however, wrapper objects, written in the native language of the framework, are required that marshal calls to the Prolog engine. In fact, from a design perspective, the approach entails two parts, so-called *stubs*. The wrapper object constitutes the first stub, while its Prolog counterpart the other. The stubs maintain a TCP or piped connection to each other through which Prolog call parameters and results are transmitted, usually as a stream of characters.

While practical in harnessing the benefits of a web development framework, this approach undoubtedly requires experience in programming both Prolog and the

```
<html logic-module="factorial">
  <h1>Factorial example</h1>
  <psp:assign var="E" expr="{atom_number(http_get(number))}">
    <psp:for-all function="between(1, E)" iterator="N">
      <li><psp:insert function="factorial(N)" /></li>
    </psp:for-all>
  </psp:assign>
</html>
```

```
:- module(factorial, [factorial/2]).
factorial(Number, Factorial) :- ...
```

Figure 4: The Prosper example document *factorial.xhtml* (above) and the Prolog module *factorial.pl* associated with it (below). Some XHTML elements (e.g. *ul*) have been omitted and full namespaces are not shown for brevity.

external encapsulating language. From a performance point of view, stubs introduce a further level of indirection into the web service chain and often operate inefficiently because the Prolog and the foreign language execution model are vastly different. Lastly, debugging Prolog embedded in foreign code is substantially harder, which can greatly increase development time.

Prosper offers a balanced mix of the two main approaches. It is a variant of the first approach in the sense that the majority of request processing and content generation is performed in Prolog or Prolog-integrated libraries. Only Prolog programming experience is required and development is eased through improved debugging. On the other hand, it is closer to the second approach in the sense that it adopts the model-view paradigm of rapid application development frameworks by splitting web applications into an application logic and a presentation layer. Application logic is coded as a regular Prolog module, while presentation is an (X)HTML document with some elements and attributes carrying extra information for Prosper to realise so-called visual transformation rules (to be explained in detail). Figure 4 shows a web page and the associated application logic that lists all factorials up to N, functionally equivalent to the web page generated by the snippet in Figure 3. Despite its verbosity, the presentation layer is not interleaved with application logic and retains its structure as a regular XHTML document. Roughly speaking, Prosper can be viewed as an extension of PiLLoW with a more robust visual front-end. Section 4 elaborates on the design of the proposed framework.

4 Architectural overview

From a design perspective, Prosper can be decomposed into two major layers (Figure 5). The lower layer, *Prolog Web Container*, either acts as a stand-alone web server or maintains a direct persistent connection to the web server through the

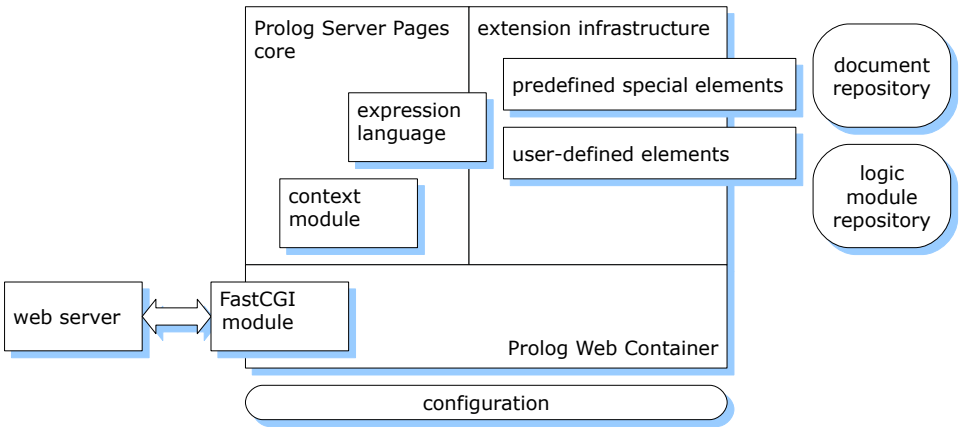

Figure 5: The architecture of the proposed framework.

FastCGI protocol. The FastCGI module transmits data to and from the Prolog framework. In addition to acting as or communicating with the web server, Prolog Web Container parses headers and payload associated with HTTP requests into Prolog terms and generates them for replies, maintains a worker thread pool and assigns jobs to threads. The primary task of the container is to isolate the communication protocol and provide a natural view of request and session data for the programmer. In accordance, the container provides similar facilities as other Prolog libraries in use, PiLLoW in particular, i.e. reversing content encoding, parsing query strings, etc.

Prolog Server Pages, built on top of the container, defines an XML-based document model. The conventional XML document model is extended with *special elements* belonging to a dedicated namespace each of which realises a *transformation rule*. A transformation rule describes how the (visual) content of an element is transformed based on attributes, and the local and global *context* of the given element. Local context corresponds to variables instantiated in server documents, while global context refers to request context as extracted by Prolog Web Container and exposed as Prolog predicates by the *context assertion* module. In assigning values to local variables, Prosper offers the so-called *expression language*. Expression language can be seen as an extension to the *is/2* predicate to include basic atom manipulation, request context variables and user-defined functions.[1]

Prosper includes a predefined set of special elements implementing the most common transformation rules such as conditionals and iteration constructs. However, the set of transformation rules is not restricted. Relying on the *extension*

[1]In this paper, a Prolog *function* corresponds to a predicate all of whose arguments are strictly inbound except for the last, which is strictly outbound, and which should be unified with a ground term and is interpreted as the return value of the function. This corresponds to the Mercury [7] definition of function.

infrastructure, the user may create new modules that contain hook predicates registered for steps associated with reply generation. Modules correspond to XML namespaces and exported hook predicate names to element names in server page documents. In fact, it is via hook predicates that the predefined transformation rules are realised in the framework, which means – in the extreme case – that they can also be redefined. Special elements and their implementor hook predicates are declared in a *configuration file*. The configuration file also holds connection settings to the web server and parallel execution parameters required by Prolog Web Container.

Apart from the visual part of Prolog Server Pages, the *logic modules* give real power to the architecture. While independent from Prolog Server Pages documents, they provide the code-behind that encapsulates true application logic as conventional Prolog modules. Server pages can reference code-behind in a variety of ways: assign server page variables based on application logic, test for the satisfiability of predicates (goals), and formulate conditions using the return value of functions, each of which may affect visual layout.

Prolog modules constituting application code reside in a dedicated directory, the so-called *logic module repository*. Similarly, Prosper maintains a *document repository*, which is the default location to search for server pages.

5 Generating a reply

In order to get a deeper insight into the internals of the framework, in this section we will trace how a request dynamically produces a reply in Prosper. As an example, let us suppose that the user has entered a URL into his browser's location bar that corresponds to a web page which lists all factorials up to 3 (e.g. *http://prosper.cs.bme.hu/factorial.psp?number=3*). Albeit the example is simple, it will illustrate the different stages of the service chain.

Once received by the web server, based on configuration settings, the server detects that this HTTP request is to be forwarded to Prosper for reply generation. It dispatches a FastCGI request, which is intercepted by one of the idle Prolog Web Container worker threads.[2] The thread extracts the context associated with the request as Prolog terms. The context typically includes query parameters in the URL (typically for HTTP GET requests), HTML form data passed as payload (typically for HTTP POST requests) and the session identifier. The Prolog representation of the context is handed over to Prolog Server Pages. In our example, the request context only contains GET parameters, represented by the list *[number='3']*.

First, Prolog Server Pages *loads* the document associated with the URL. The loaded document is *preprocessed* into a so-called intermediate term (IT) representation. Context is then *asserted* into a dedicated module and the document is *evaluated*. Evaluation ends with generating *output*, which is returned by Prolog Web Container to the web server as a FastCGI reply (Figure 6). So-called *transformation rules* are associated with both the preprocessing and the evaluation phase.

[2] See predicate *worker/1* in module *prosper_server* [13].

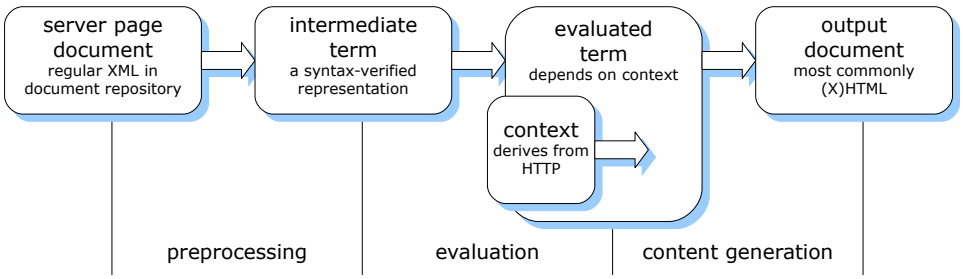

Figure 6: The phases of reply generation for server page documents that have not been cached.

The crucial difference is that in the preprocessing phase, no external context information is available, while evaluation-time transformation is context-sensitive. The aforementioned steps are elaborated below.

Loading a Prosper document. The role of the loading phase[3] is to fetch a referenced document from disk and construct its Prolog XML term representation, similar to the one used by the PiLLoW library [5].

Whenever an HTTP request corresponds to a server page that has not been loaded, Prosper looks for the page in the document repository. Let us suppose that the URL entered by the imaginary user does not correspond to a loaded document. Therefore, the document *factorial.xhtml* (Figure 4) is loaded and parsed into a Prolog XML term representation as seen in Figure 7. This representation mainly consists of nested *element/3* terms, where the arguments represent:

1. the name of the XML element after namespace resolution;
2. a list of attributes associated with the element;
3. a list of nested XML nodes as *element/3* terms for XML elements or atoms for character data.

Preprocessing phase. The goal of the preprocessing phase,[4] the next link in the service chain, is to validate the loaded document. Preprocessing ensures that special elements referenced by the document exist, they are used correctly in terms of syntax, and that the logic module associated with the document is loaded and compiled.

As previously mentioned, special elements correspond to transformation rules. What the transformation rule exactly does depends on the attributes associated with the element and its context. In our example, *psp:assign*, *psp:for-all* and

[3]Implemented in *import_page/3* in module *prosper_core* [13].
[4]Implemented in *markup_to_term/6* in module *prosper_core* [13].

```
elem(html, ['logic-module'=factorial], [
  elem(h1, [ ], ['Factorial example']),
  elem(psp:assign, [var='E', expr='{atom_number(http_get(number))}'], [
    elem(psp:for-all, [function='between(1, E)', iterator='N'], [
      elem(li, [ ], [
        elem(psp:insert, [function='factorial(N)'], [ ])
      ])
    ])
  ])
])
```

Figure 7: The XML term representation of the example document in Figure 4.

psp:insert are special elements, assuming the namespace *psp* is registered with Prosper.[5] The *psp:assign* special element can have a *var* attribute, which specifies the name of the variable to introduce in the scope of the element. Similarly, *psp:insert* is used with the attribute *function* in the example to insert a return value but could also be used in conjunction with *expr* to insert the value of an expression.

However, in the preprocessing phase no context information associated with the HTTP request is available; it has not yet been asserted. In spite of this, verifying attributes, parsing atoms into terms, etc. are already possible. These operations are performed by *preprocessing-time hooks* for each special element. A hook predicate interprets element attributes and/or contents and has the following signature (*elementName* denotes the name of the special element without the namespace):

elementName(+VarTypes, +Attrs, +Contents, -Terms)

Here, *VarTypes* propagates type information,[6] *Attrs* is a list of *Name=Value* pairs, which consists of attributes that parametrise the element. *Contents* is a list of inner elements in XML term representation. *Terms* is the single output argument of the predicate, which is the IT representation (preprocessed form) of the element and is commonly bound to a single-element list of the following form:[7]

[extension(ModuleName:Predicate, ContentTerms)].

In this term, *ContentTerms* has similar semantics as *Terms* in the enclosing element: it is the IT representation of the enclosed child elements. This suggests a recursive way of operation. Indeed, albeit not compulsory, most special elements compute their own IT representation based on that of their descendants. *Predicate* corresponds to a Prolog predicate, which (augmented with some additional arguments) will be called in the evaluation phase to generate output. In other words,

[5] For conciseness, namespaces are not written out as full URLs, even though in the actual implementation, they are used in that manner.

[6] Although there is no strict typing scheme in Prolog, some degree of type enforcement allows catching errors at an earlier stage.

[7] In fact, *Terms* is a list of atoms, *element/3* and *extension/2* terms. However, only *extension/2* terms are subject to evaluation in a later phase thus *Terms* is usually a list with a single *extension/2* element.

```
elem(html, [ ], [
  elem(h1, [ ], ['Factorial example']),
  extension(assign_expression('E', EL), [
    extension(for_all(factorial:between(1, E)-['E'=E], 'N'), [
      elem(li, [ ], [
        extension(insert_function(factorial:factorial(N)-['N'=N]), [ ])
      ])
    ])
  ])
])
```

Figure 8: The intermediate term representation of the example document. *EL* denotes the execution plan of the expression language term and is omitted for conciseness.

Predicate is the evaluation-time transformation rule associated with the special element parametrised with *Attrs*. For instance, a different *Predicate* is associated with a *psp:assign* element if it assigns a variable based on an expression than if based on a function call. In fact, arguments present in *Attrs* as an association list are converted into positional arguments with appropriate conversions where necessary (e.g. atoms converted to Prolog goals).

Hook predicates should never fail but should signal malformed syntax (such as an unrecognised attribute) by throwing an exception.[8] This guarantees that the document is syntactically well-formed at the end of the preprocessing phase.

In order to better comprehend the preprocessing phase, we compare the XML term representation of our example document in Figure 7 with its preprocessed version in Figure 8.

In the case of the root element *html*, the two representations are identical, except for the attribute *logic-module*. This attribute binds a conventional Prolog module (application logic or presentation code-behind) to the Prosper document. The exact location of the module source file is either directly specified in the *logic-module* attribute as an absolute path, or it may be a relative path, in which case it is searched for w.r.t. the module repository. Any predicates that occur in the document are auto-qualified with the name of this module during the preprocessing phase.

The first notable difference is *psp:assign*, which has been converted into an *extension/2* term. *assign_expression* is the name of a predicate that computes an expression language (EL) term and assigns its value to a (server page) variable. The scope of the variable is the enclosed contents of the *psp:assign* element. The function *http_get* in the EL term returns the string value of a query string variable, while *atom_number*, as its name suggests, converts its operand to a Prolog number. Just as documents, EL expressions are preprocessed, yielding an *execution plan*,

[8] The rationale behind throwing an exception is that simple failure prevents extracting the context of the actual error, which is necessary in order to print the proper error message.

which is not shown in Figure 8. The execution plan is a compound term that contains

1. the uninstantiated variables in the expression, and
2. the module-qualified names of the Prolog functions to call to compute the result.

The representation of the special element *psp:for-all* has also changed substantially. The atom in its attribute called *function* has been converted into a real Prolog term augmented with a list of uninstantiated variables in it. *for_all(Function-Insts, Variable)* is a predicate that instantiates variables in *Function* and calls it, returning results in a local variable. Subsequent solutions are obtained through backtracking. Note the number of arguments to *between/3* (the third, output argument is absent and is appended automatically) and the auto-qualification.

For the sake of higher performance, Prosper caches preprocessed documents. If a document is available in the cache, the loading and preprocessing phases are skipped.

Request context assertion. Context information available in Prosper documents and logic modules is loaded in the request context assertion phase. The primary goal of this phase is to expose HTTP request context (such as request parameters and session variables) to EL functions and logic module predicates in a natural manner without having to propagate an extra argument encapsulating the context. Predicates in the module *psp* store context information by means of *thread-local* blackboard primitives [15]. Whenever a mutable (session) value is modified while the request is served (e.g. a session variable is assigned to), changes are recorded in a (thread-global) dynamic fact database at the end of the subsequent evaluation phase. Hence, no particular thread is associated with any session and any worker thread may serve any request. Worker threads load current values from the dynamic fact database into blackboard primitives before the evaluation phase and record new values when evaluation ends.

Logic modules have access to context by calling predicates exported by the module *psp*. For instance, the *http_get/2* and *session/2* predicates retrieve the value of a GET and a session variable, respectively, and the predicate *session_set/2* assigns a value to a session variable.[9] For maximum conformance to the Prolog execution model, they all support backtracking, i.e. assignments to session variables are undone upon failure in a logic module predicate.[10]

Evaluation phase. In the last major phase, *evaluation*,[11] the preprocessed document is transformed w.r.t. the available request context. By the end of the evaluation phase, the document has been transformed into a term representation whose string equivalent is ready to be sent back directly to the client as response.

[9] For a full list, see exported predicates in module *psp* [13].
[10] SWI-Prolog provides backtrackable destructive assignment on blackboard primitives.
[11] Implemented in *term_to_elements/3* in module *prosper_extensions* [13].

From a declarative point of view, each IT element represents an (evaluation-time) transformation rule, influenced by

1. term contents,
2. asserted HTTP request- and session-related data that is globally accessible in the entire document, and
3. local variables assigned by outer special elements (i.e. that encapsulate the element to which the rule corresponds) such as *psp:assign*.

In the case of *element/3* terms, the (recursive) transformation rule is trivial: transformation rules are applied to each child element with the same context as for the parent element and the evaluated form of the parent element comprises of the combined results of these transformation rules. For *extension/2* terms, recall that the first argument corresponds to a hook predicate assembled in the preprocessing phase: this is what represents the transformation rule. From a procedural point of view, in fact, the IT representation is traversed top-down, at each depth invoking hook predicates or the trivial transformation rule, where hook predicates may introduce new local variables before processing the children of the term they correspond to.

Local variables are means to store and reuse calculated data within server page documents. In contrast to globally available data (loaded into the thread-local module *psp* in the context assertion phase), they are accessed as Prolog variables rather than predicates and they are confined to the server page document in which they are introduced and may not be directly used in presentation code-behind or application logic files. More precisely, the scope of local variables is always restricted to the descendants of the element in which they are assigned and are hidden by variables of the same name. Server page local variables have similar semantics as Prolog or XSLT variables in the sense that they can be assigned only once. Contrary to Prolog, however, variables cannot remain uninstantiated and are unified immediately in the element in which they are introduced.

Figure 9 shows the evaluated version of the preprocessed document in Figure 8. The result should not be surprising. For the elements *html*, *h1* and *li*, the trivial transformation rule has been applied and they are intact except for their recursively processed contents. The IT equivalents of special elements *psp:assign*, *psp:for-all* and *psp:insert* are absent from the output but their effect is apparent. The local variable *E*, which is introduced by *assign_expression*, has been used to instantiate unbound variables in the function *between(1, E)*, and the iteration variable *N* of *for_all* has been used multiple times to call the function *factorial(N)*. *N* behaves as expected, taking a different value for each loop of the iteration.

From the perspective of the framework, local variables are in fact *Name=Value* members in an association list. The association list is initially empty for the root element but may be extended with further members by any transformation rule, in which case the recursively processed descendant elements see the extended list. In our example, the evaluation-time transformation rule associated with the *psp:assign*

```
elem(html, [ ], [
  elem(h1, [ ], ['Factorial example']),
    elem(li, [ ], ['1']),
    elem(li, [ ], ['2']),
    elem(li, [ ], ['6'])
  ])
])
```

Figure 9: The evaluated form of the example document.

element prepends the variable E to the name-value list, while the rule related to *psp:for-all* does so with N.

6 Implementation

Prosper is implemented mainly in SWI-Prolog and partially in C. The most notable SWI-specific extra services utilised by the framework are XML document parsing and generation, blackboard primitives, multi-threading and basic thread communication.

The framework comprises of the following major components:

1. The *server module* implements Prolog Web Container.
2. The *core module* manages the lifecycle of a Prosper page. In particular, it imports pages on demand, initiates context assertion, page preprocessing and evaluation, and outputs error documents.
3. The *context module* asserts and retracts thread-local data via blackboard primitives to expose request, session and user preference values, all of which are manipulated through dedicated predicates of the module *psp*.
4. The *extension module* contains predicates essential to special element implementors. It includes helper predicates to aid XML attribute parsing and the predicates *element_to_terms/3* and *term_to_elements/3*, which realise page preprocessing and evaluation, respectively. The latter two predicates are called by transformation rule hooks to recursively process child elements.
5. The *built-in elements module* contains the predefined set of special elements, including simple and compound conditionals, iteration constructs, variable assignment and insertion.
6. The *expression language module* is responsible for expression language execution plan generation and expression evaluation.
7. The *FastCGI foreign language module*, written in C, implements the FastCGI protocol.

Table 1: Comparative performance of various frameworks.

Development tool	Application model	small	large	intensive
SICStus Prolog 3.12.5	CGI, saved state	165.78	225.33	n.a.
SWI-Prolog 5.5.33	CGI, saved state	39.47	60.17	n.a.
PrologBeans.NET	ASPX	6.297	7.781	91.91
Prosper (PWC + PSP)	multi-threaded FCGI	2.688	8.719	5.828
Prosper (PWC only)	multi-threaded FCGI	1.938	6.953	n.a.
SWI-Prolog 5.6.27	standalone server	1.266	6.313	n.a.
static html content		0.875	1.406	n.a.

While primarily designed to increase designer and programmer effectiveness, the proposed architecture is comparable to other Prolog-based technologies in terms of speed. In a loopback scenario (i.e. server and client were running on the same machine), different configurations were polled by HTTP requests with GET parameters. All configurations parsed the query string, computed a simple arithmetic expression based on query parameters, and displayed results in a web page. CGI and FastCGI-based applications (Prosper inclusive) connected to Apache/2.0.54, .NET applications ran on the built-in web server provided with Visual Studio 2005. Benchmarking was performed by ApacheBench 2.0.41 on an AMD Athlon64 3000+ running Microsoft Windows XP Professional SP2.

Table 1 shows cumulative response times in seconds for 1000 requests with 2 concurrent threads. In test cases *small* and *large*, responses of about sizes 1kB and 50kB were requested with few embedded Prolog calls. In test case *intensive*, the architectures had to call about 50 Prolog predicates in application logic to produce a result of about 3kB in size. n.a. indicates that there is no overhead of a Prolog call-intensive setup (i.e. presentation and application logic are not separated and both are in Prolog) or it is not meaningful for the test case. Static HTML content is included for reference, and is meant to indicate the absolute lower bound for response time as (unlike the other scenarios) it requires no extra overhead owing to context evaluation.

Three cases are of special interest. The standalone multi-threaded HTTP server shipped with SWI-Prolog can serve as the basis for comparing the performance of Prolog-based frameworks. It provides convenience tools for HTTP reply generation but intermixes presentation and application logic. The difference in speed between Prosper with Prolog Web Container and SWI's standalone server gives an estimate of the cost of using an intermediary FastCGI transmission. The extra overhead of Prosper with the Prolog Server Pages document model shows the relative cost of having a separate presentation and application logic layer. The careful reader may also notice that Prosper is slower than PrologBeans.NET when large documents with few Prolog calls are served. This is attributable to the greater efficiency ASP.NET handles strings than Prolog handles atoms in a multi-threaded environment.

7 Summary, perspectives for future work

In this paper, a framework that facilitates developing web-oriented Prolog applications has been presented. With a persistent multi-threaded architecture, an XML-based document model and a set of reusable transformation rules, it provides an efficient yet convenient way to create web applications in Prolog. Code changes required in existing Prolog modules for the sake of web presentation are minimal and web pages constituting the presentation layer can be composed with a declarative way of thinking in any arbitrary XML editor. Presentation and application logic are clearly separated, thus application logic can be debugged and maintained independently. Lastly, the framework integrates well in existing web server scenarios and is open to extension.

As seen in the factorial example, the current implementation of Prosper generates content by traversing a term that constitutes some transformed version of the presentation source document. Compilation of these source documents into Prolog predicates could contribute to increased performance, especially in the case of long documents that have considerably large tree equivalents.

Besides dynamic content generation based on Prolog application code, web services offer another approach to integrate Prolog into complex information systems. As Prosper provides a straightforward way to parse and generate markup content, consuming and producing SOAP envelopes corresponding to Prolog calls and solutions seems a natural future extension.

Acknowledgements

The author acknowledges the support of the Hungarian NKFP Programme for the SINTAGMA project under grant no. 2/052/2004.

References

[1] Achour, Mehdi, Betz, Friedhelm, Dovgal, Antony, Lopes, Nuno, Richter, Georg, Seguy, Damien, Vrana, Jakub, et al. *PHP Manual*. PHP Documentation Group, 2007. `http://www.php.net/manual/en/`.

[2] Armstrong, Eric et al. *The J2EE 1.4 Tutorial (For Sun Java System Application Server Platform Edition 8.1 2005Q2 UR2)*. Sun Microsystems, June 2005.

[3] Bray, Tim et al., editors. *Extensible Markup Language (XML) 1.0*. World Wide Web Consortium, 4th edition, August 2006. `http://www.w3.org/TR/2006/REC-xml-20060816/`.

[4] Brown, Mark R. *FastCGI specification*. Open Market, Inc., April 1996. Document Version: 1.0.

[5] Cabeza, Daniel and Hermenegildo, Manuel. *The PiLLoW Web Programming Library*. The CLIP Group, School of Computer Science, Technical University of Madrid, January 2001. `http://www.clip.dia.fi.upm.es/Software/pillow/pillow.html`.

[6] Fielding, R. et al. *Hypertext Transfer Protocol – HTTP/1.1*. Network Working Group, The Intenet Society, June 1999. RFC 2616.

[7] Henderson, Fergus et al. *The Mercury Language Reference Manual*. University of Melbourne, 2006. Version 0.12.2.

[8] Hunyadi, Levente. Prosper: A framework for extending Prolog applications with a web interface. In Dahl, Verónica and Niemelä, Ilkka, editors, *Proceedings of the 23rd International Conference on Logic Programming*, Logic Programming, pages 432–433, Porto, Portugal, September 2007. Springer. LNCS 4670.

[9] Hurwitz, Dan and Liberty, Jesse. *Programming ASP.NET*. O'Reilly, 3rd edition, October 2005.

[10] Johnston, Benjamin. Prolog server pages. `http://www.benjaminjohnston.com.au/template.prolog?t=psp`, 2007.

[11] Nuzzo, Mauro Di. Prolog Server Pages: A server-side scripting language based on Prolog. `http://www.prologonlinereference.org/psp.psp`, April 2006.

[12] Pemberton, Steven et al. *XHTML 1.0 The Extensible HyperText Markup Language*. World Wide Web Consortium, 2nd edition, August 2002.

[13] Prosper hosted at SourceForge.net. `http://prospear.sourceforge.net/`.

[14] Swedish Institute of Computer Science. *SICStus Prolog Users Manual*, May 2007. Release 4.0.1 `http://www.sics.se/sicstus/docs/latest/html/sicstus/PrologBeans.html`.

[15] Wielemaker, Jan. Native preemptive threads in SWI-Prolog. In Palamidessi, Catuscia, editor, *Practical Aspects of Declarative Languages*, pages 331–345, Berlin, Germany, December 2003. Springer Verlag. LNCS 2916.

[16] Wielemaker, Jan and Anjewierden, Anjo. An architecture for making object-oriented systems available from Prolog. In *WLPE*, pages 97–110, 2002.