Prolog Server Faces – A Declarative Framework for Dynamic Web Pages

Christian Schneiker[1], Mohamed M. Khamis[2], and Dietmar Seipel[1]

[1] Department of Computer Science,
University of Würzburg, Am Hubland, D – 97074 Würzburg, Germany
{christian.schneiker|dietmar.seipel}@uni-wuerzburg.de

[2] Department of Computer Science,
German University in Cairo, Egypt
mkhamis89@gmail.com

Abstract. With Prolog Server Faces, we provide a stateful and event driven framework for dynamic web applications written in PROLOG and XML. Following the MVC concept, the *view* of web pages is fully specified in a compact XML definition with statements for processing backend logic in PROLOG. Our framework provides an extensive, and easy to extend, tag library for compact XML, which will be expanded to XHTML with AJAX support, and an HTTP server implementation for the backend logic processing. Moreover, it is possible to use existing JSF-XML files with PSF.

1 Introduction

In the last years, web applications have been a breakthrough in client-server computing: they are *cross-platform compatible*, since they operate through a web browser, they require *very little disk space* on the client side, and they can be *upgraded* and *integrated* with other web procedures easily. A web application takes an HTML website from static pages that only display content to interactive, dynamically generated HTML web pages.

The HTTP protocol is very simple and mostly TCP–based: the client sends a request, the server replies with a response. Both request and response are text-based messages, each message contains a header, and sometimes a body. The message exchange is done without saving or storing any information, making it a stateless protocol.

HTTP excels at delivering static websites. For interactive web applications, however, it will be tedious and tiring to parse headers, understand them, then reply in another header following the required format. For this reason, the need of server pages arose: server-side code is stored on the web server; when a client requests a dynamic page, the request is parsed, the requested page is processed, and the server replies with an HTML page, that can be understood by the client; these are pages generated by the server-side code, which can change dynamically according to the web application's needs. Several technologies have been introduced for server-side scripting, such as ASP, PHP, and JSP.

Scripting languages are easy to learn, and they can be written in the same files with the HTML, with interleaving HTML and scripting code. However, this flexibility comes at the expense of a well-designed application, maintainability, security, and sometimes it is slower, because the code is interpreted and not compiled. This is where frameworks come into the picture: they facilitate the development process, and they make the code neater and better organized.

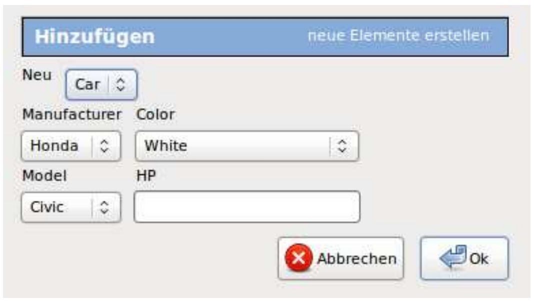


Figure 1. A web dialog generated from a database table and foreign key constraints.

The aim of this paper is to use PROLOG for server side coding by implementing Prolog Server Faces, a server faces technology that uses XML-based tag libraries [3, 16]. The library elements are transformed into standard XHTML pages using the PROLOG library FNQUERY for querying and transforming XML documents and data containers [12, 13]. JavaScript methods using AJAX with PROLOG predicates are implemented. An HTTP web server is implemented using an HTTP library for SWI-PROLOG. PSF separates the front-end and back-end, making design patterns like *Model-View-Controller* (MVC) or *Facade* easily viable.

The paper is structured as follows: the next section gives a short overview of the known web frameworks for implementing stateful applications in Java, followed by the approaches of combining PROLOG with web applications. Section 3 describes our implemented Prolog Server Faces technology. It shows the transformations of XML elements to write short and reliable code. We will also discuss how to use AJAX for combining XML with nested PROLOG statements, as well as the integration with relational databases. Section 4 explains PSF with the help of a case study, that shows the easy implementation of applications according to the MVC concept. The last section gives a short conclusion and shows possible future work.

2 Related Work

The following section describes JavaServer Faces, a Java EE web framework based on *servlets* and JSP technology, for developing dynamic web pages with object-oriented backend engines [6, 7, 15]. We will also talk about on Prolog Server Pages, an approach that combines HTML with PROLOG for web-based scripting, and explain two different techniques which have been developed over the last years [5, 14]. The combination of functional logic programming and web interfaces has been discussed in [4].

In general, with Server Faces it is possible to separate the *view* of web applications from their *model* and *controller*, according to the MVC concept. Server Pages, in contrast, just allows the developer to implement the *controller* within the HTML code, as with common web scripting languages like PHP.

2.1 JavaServer Faces (JSF)

As a framework for server-side user interface components, Sun Microsystems and other companies initially released JavaServer Faces in 2004. JavaServer Faces use XML for implementing the view of web pages according to the MVC concept. In contrast to static HTML pages or JSP, JSF provides *stateful* web applications, *page templating* or even AJAX support and gives the ability to develop server applications within the object-oriented programming language Java.

JSF allows processing client-generated events, to alter states of components, making them event-oriented. It includes *backing beans*, which synchronize Java objects to user interface components. Unlike desktop programs, web-based applications are expected to be accessed from different client types, such as desktop browsers, cell phones, and PDAs, JSF provides a flexible architecture allowing it to display components in altered ways, and it also offers many validation techniques.

As a server-side technology, all pages requested by the client are preprocessed by the server. Via HTTP, every requested XML document is transformed to standard XHTML, and nested calls to Java objects, which are specified in an *expression language*, are processed. The following example shows a JSF-XML element, which is transformed to standard XHTML. The `selectOneMenu` element has an additional attribute `value` with a Java expression for setting the right value, which is read from a data container, a Java Bean.

```
<h:selectOneMenu id="selectCar" value="#{carBean.currentCar}">
   <f:selectItems value="#{carBean.carList}" />
</h:selectOneMenu>
```

In this example, a list of cars is read from the Bean and according to the values, a set of `option` elements is generated. The `selectOneMenu` element is transformed to a normal XHTML `select` element, and necessary attributes like `name` and `id` are added. The resulting valid XHTML page is transferred to the client and rendered by a browser.

```
<select id="selectCar">
  <option value="corolla">Corolla</option> ...
</select>
```

The framework uses standard Java classes to transform the documents with common Java component tree operations. Even when the work with object-oriented programming languages and XML tree operations is hard to read and to debug, it makes it possible to extend the core libraries for the transformations (*core taglib*) by writing new classes and adding them to the library.

2.2 Prolog Server Pages (PSP)

For combining the features of logic programming and web based applications, some approaches have been developed over the last years. These PROLOG and HTML scripting techniques allow the implementation of *dynamic web pages*; nevertheless it is not possible to separate the program logic from the user interface and it makes the code hard to read. In this paper we want to discuss two major implementations of Prolog Server Pages, a technique very similar to JavaServer Pages, which allows inline PROLOG scripting in HTML documents.

PSP Chunk Programming. In the first PROLOG implementation of Server Pages, a programmer has to write HTML elements within the PROLOG source code files while chunks of PROLOG code are encapsulated like `<?psp Chunk ?>` [14]. A chunk can consist of a sequence of PROLOG rules followed by a sequence of PROLOG directives issuing PROLOG goals. This implementation forces the developer to call `write` predicates whenever something is desired as an output, even the HTML elements. Standard PROLOG goals are just interpreted as usual. Additionally, the querying is mixed with the declaration; this makes design and maintainability harder, and the developer will have to duplicate the code if a PSP chunk needs a predicate defined in a previous chunk.

The PSP server passes HTML tags to standard output and interprets PROLOG code within the PSP elements in the standard way of PROLOG. In the following `Hello World` example, the predicate `greeting_message` defines the string `'Hello World!'`. The directive starting with `?-` issues a query which binds `'Hello World!'` to the variable `X` and writes it to standard output.

```
<html> <body>
   <?psp
      greeting_message('Hello World!').
      ?- greeting_message(X), write(X).  ?>
</body> </html>
```

The result is a standard HTML document.

```
<html><body>Hello World!</body></html>
```

PSP with General Server Pages. In the second implementation by Benjamin Johnston, the aim of the Prolog Server Pages web-based scripting language was to implement dynamic web applications using PROLOG, avoiding manual parsing of common HTML elements within the source code [5].

The syntax for combining HTML and PROLOG scripting elements in this implementation is comparable to that of PHP, ASP and JSP, using tags like `<?,` and `,?>`, which is standardized in the General Server Pages approach. This allows having HTML and PROLOG code in the same file; however, this might come at the expense of design problems, especially if the developer wants to maintain a design pattern throughout the structure of his web application.

The following is another Prolog Server Pages example for `Hello World`. The predicate `greeting_noun` holds the string `World`, and it is defined outside of the HTML section. The begin and the end of the HTML code have to be marked with `/*` and `*/`, respectively, which is necessary for the PROLOG compiler and treats the element like normal comments. Within this block, it is now possible to write PROLOG goals within `<?, Goal ,?>` which the server will execute. Here `X` is bound to `'World!'`. The result can be written to standard output within `<?= Term ?>` tags. If `Term` is bound, then its value is written, otherwise just the word `Term` is written to standard output. In this example, the string `'Hello '` is written followed by `'World!'` from the PSP code.

```
greeting_noun('World!').
/*
<html> <body>
   <?, greeting_noun(X) ,?> Hello <?= X ?>
</body> </html>
*/
```

The result generated by the PROLOG server is similar to the previous example.

2.3 FNQUERY and FNTRANSFORM

For the transformations in our framework, we extensively use the XML query, transformation and update language FNQUERY [12, 13], which is fully interleaved with SWI-PROLOG. Like with XPATH, it is possible to query complex structures with path expressions and axes. As an extension of XPATH, it is possible to select branches over deeply nested structures. The sublanguage FNTRANSFORM, which extends XSLT, allows to transform XML elements in PROLOG using normal syntax.

FNQUERY uses triples for representing XML documents. E.g., for the association list `As = [color:red, model:civic]` of attribute/value pairs, `cars:As:Es` represents an XML element with the tag `cars`; the content `Es` can be a (possibly empty) list of such triples.

The path language FNPATH of FNQUERY is very similar to XPATH. Compound terms with the functor `/` are used for selecting subelements of an element. The functor `@` is used for selecting attribute values. E.g., the binary predicate `:=` in the call

```
?- M := doc(cars.xml)/car@model.
```

selects the value for the attribute `model` from the element `car` in the XML document `cars.xml` below and binds the result to `M`.

```
<cars>
   <car id="corolla" model="Corolla" />
   <car id="civic" model="Civic" />
   <car id="city" model="City" />
</cars>
```

It is even possible to query with *multiple* location paths. The following expression selects the attributes `id` and `model` and forms pairs `[Id, M]` of the results:

```
?- Pair := doc(cars.xml)/car-[@id, @model].
```

The library FNTRANSFORM is used for implementing transformations. In Section 3.1, we will use calls `X ---> Y` for transforming FN triples `X` to other FN triples `Y`.

3 The Framework Prolog Server Faces (PSF)

PSF is a *stateful* and *event-driven* framework that integrates logic programing in modern web applications. We are combining the different PSP approaches described in Section 2.2 for mixing common PROLOG with XHTML to develop *dynamic* web pages with the advantages of JSF for writing *condensed* XML. This XML will be expanded to normal XHTML with connection to XML documents and relational databases for data handling. We provide an application programming interface for combining an extended HTTP server implemented in SWI-PROLOG with a huge and easy to extend tag library for defining web pages in a compact XML structure. For the transformations of XML elements, we use FNTRANSFORM.

3.1 Standard PSF Transformations

Like in JSF, nearly every XHTML element can be written in a compact form with additional attribute values, which read the data from complex data structures like term structures, XML documents, or even relational databases. In our PSF framework, we have implemented the *core tag library*, which consists of tags like HTML `form`, the different `input` element types, and of course `radio` buttons and `select` menus.

We want to exemplify the work with PSF-XML files with the following code of a single select menu, whose data are stored in an additional XML file. The PSF-XML page contains only two elements for defining the type of the select menu as well as an element with an FNPATH expression, which handles the data for the different option types, in this case the different car models.

```
<h:selectOneMenu id="selectCar">
   <f:selectItems value="#{doc(cars.xml}/car-[@id, @model]}" />
</h:selectOneMenu>
```

The data can be read from either an XML document or from PROLOG data structures. The transformation itself is handled by FNTRANSFORM, which is integrated in our framework. When a client requests such a file, the server automatically transforms it to XHTML with one of its request handlers. The following code shows such a transformation from `selectOneMenu` to `select` elements:

```
X ---> Y :-
   X = 'h:selectOneMenu':As_1:[Item],
   Y = select:As_2:Items.
%  attributes
   ( Id := X@id ; Id = '' ),
   As = [id:Id, name:Id, size:1],
   fn_association_lists_union(As, As_1, As_2),
%  subelements
   ( Expression := Item@value ; Expression = '' ),
   psf_evaluate_expression(Expression, Pairs),
   ( foreach([V, M], Pairs), foreach(I, Items) do
        I = option:[value:V]:[M] ).
```

Firstly, the attribute list is extended by the attributes `id`, `name`, and `size`; if these were already present, then the old values are kept. The `id` is taken from `X`; if `X` does not have an `id`, then it is set to the empty string as a default value. Secondly, the `option` subelements are generated based on the path expression in the attribute `value` of `Item`. In our example, the list `Pairs` given by

```
[[corolla, 'Corolla'], [civic, 'Civic'], [city, 'City']]
```

is derived, since the path expression selects the attributes `id` and `model` of the car elements in the file `cars.xml`. Finally, each pair `[V,M]` yields an `option` subelement `I`. FnTransform works bottom-up, and there is no transformation rule for `selectItems` elements. Thus, these elements remain unchanged first. However, depending on the context – in our case `selectOneMenu` – they are transformed to other elements.

The output of the transformation is valid XHtml code, which can be rendered by the browser to a select menu with the different option elements.

```
<select id="selectCar" name="selectCar" size="1">
   <option value="corolla">Corolla</option>
   <option value="civic">Civic</option>
   <option value="city">City</option>
</select>
```

3.2 Database Support

Web interfaces are often connected with a database. Therefore, we have extended the valid lists of Psf attributes to specify the additional `type` of elements; here, `type` is set to `dialog` to generate a user dialog automatically from the database structure. In such a case, an attribute `value` defines the database name we want to connect to and the tables needed for the dialog definition.

For a dialog like in Figure 1, the transformation generates a select menu with the table names of the database tables specified in the attribute `value`. Each different selection of one of these tables in the select menu forces the Prolog server to read the database schema and automatically generate a form element with different input types according to the schema; normal attributes result in single text field for inputs, foreign key constraints construct other select menus. For these foreign key select menus, the referenced tables are read and only valid values are set to the menu; the user cannot enter wrong data. Of course, it is also possible to select further values from the referenced table, other than just the different foreign key values, and to display them in the menu to make the generated dialog more readable.

3.3 Ajax-Based User Interaction

To implement the *controller* – the backend logic of the Prolog Server Faces – we need to handle user interactions from the web interface. In Psf, it is possible to use Ajax by calling Prolog predicates from JavaScript. Psf comes with some predefined JavaScript functions, which can easily be included in the Xml document with a common `script` element. The two main functions for combining native Prolog with

JavaScript and AJAX are `sendRequestPL(arg0, arg1, ..., argN)` for sending values from `form` elements to the server and `sendRequestXML(arg0, arg1, ..., argN)` for complete XML elements. The argument `arg0` is the PROLOG predicate to be called by the server. The subsequent arguments `arg1, ..., argN-1` are the parameters. The last argument `argN` specifies the XML id, which is refreshed with AJAX after the servers send the response. The second JavaScript function sends complete XML elements to the server. Similar to the JavaScript function above, the first argument is the predicate to be called, while the last argument is the XML id to be refreshed.

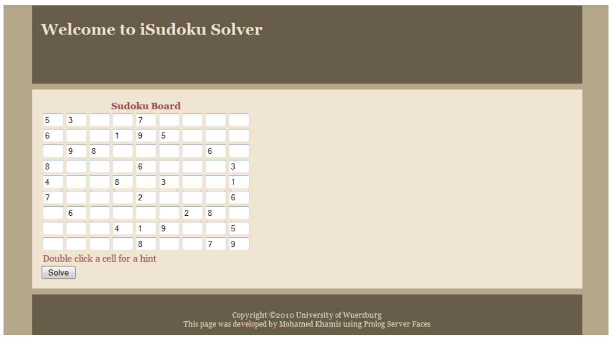
For transmitting the different parameters, we use special XML envelopes. E.g., a message of the type `send` is used for sending a predicate with its parameters:

```
<message type="send"> <predicate>...</predicate>
   <parameter>...</parameter> ... <parameter>...</parameter>
</message>
```

The server processes the message and responds with a newly generated XHTML element, and the browser can now update using a JavaScript `xhr` function.

4 Using the MVC Concept for a Sudoku Solver

We have implemented a Sudoku solver based on PSF and two different open source implementations of the backend logic in PROLOG and CLP, respectively. Although the application can also be developed with JSF, with PSF it is possible to benefit from logic programming, and it is possible to change the backend logic during runtime.



Welcome to iSudoku Solver

Sudoku Board

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Double click a cell for a hint

Copyright ©2010 University of Wuerzburg
This page was developed by Mohamed Khamis using Prolog Server Faces

Figure 2. A sudoku solver web application developed with Prolog Server Faces

According to the MVC concept, the implementation will be devided into three main parts. The *model* holds the default values of the different text fields of the user interface, and it is updated during each processing step for storing the entered values. On page load, the data container is loaded, and the initial values are compiled into the generated XHTML web page. Each element can be transformed using FNTRANSFORM in PROLOG or even in PSF-XML elements.

The second part is the *view*: the graphical user interface of the application. It is a well-formed PSF-XML page with the regular elements; the body has few PSF elements, which will be expanded to XHTML during the transformation. We use different namespaces – like h and f – to distinguish them from the regular (X)HTML.

```
<h:form>
   <f:tableGrid columns="9" rows="9"
      value="#{doc(sudoku_data.xml)/cell@value}" >
      <h:inputText size="1" ondblclick="sendRequestPL(
         'sudoku_hint', this.id, this.id)"/>
   </f:tableGrid>
   <input type="button" value="Solve"
      onclick="sendRequestXML('solve_sudoku', 'view', 'view')"/>
</h:form>
```

This PSF code will generate a table grid with 81 text fields, like in Figure 2, the values are imported from the XML container mentioned above by providing an expression in the attribute value, or if it is not desired to do so, one can exclude the attribute. The XHTML document which is generated from the PSF-XML file consists of more than 400 lines of code; thus, PSF makes it possible to generate complex web pages from short and compact XML code.

As it can be seen from the implementation, it is easy to use PROLOG to solve problems and puzzles, and with PSF, it is possible to have a neat interface, and even a web application. At the same time, PSF preserves well-formed code that can be logically divided into *Model*, *View* and *Controller* layers, which makes maintenance much easier.

The usage of MVC proves more powerful than an extra layer for the solver, and the implementation of the solver can be changed with very little work of integration. Since we have decided to use the JSF syntax, it is possible use the same XML documents for JSF and PSF; only the AJAX calls to Java methods have to be changed.

5 Conclusions and Future Work

We have introduced Prolog Server Faces, a framework for *stateful*, *event-driven* web application with AJAX support and PROLOG backend logic. Our concept is fully integrated in SWI-PROLOG, and it provides a huge *tag library* for XML element transformations from PSF-XML to standard XHTML. The tag library can be easily extended to fit the developer's needs to implement reliable and easy to read user interfaces. We have also introduced methods for combining the stateless XHTML pages with XML documents for storing data or even accessing internal PROLOG term structures or databases. While PSF uses the same XML elements as JSF, it is possible to use already developed JSF interfaces and enhance them with the power of PROLOG and CLP.

PROLOG is a good choice when there is an aspiration for a short and concise code. PSF has added an interface for this powerful language: in addition to making it applicable on the Internet, PSF makes it possible to use PROLOG engines in web applications. Following the MVC concept, the separation of the *view* and the *backend logic*, the controller can be changed easily, even during runtime of the web application.

In another project, we are combining the XUL framework [9] with SWI-PROLOG. A next step is to automatically convert user interfaces between these two technologies for providing a platform-independend framework for graphical applications in the field of logic programming. It is even possible to parse natural text and generate the PSF-XML structure for the web interface automatically [10]. Future work will consider adding further functionality to PSF, such as cookies and session management predicates, and even developing validation tools.

References

1. BÖHM, A., SEIPEL, D., SICKMANN, A. ,WETZKA, M.: SQUASH*: A Tool for Designing, Analyzing and Refactoring Relational Database Applications*. Proc. 17th Intl. Conference on Applications of Declarative Programming and Knowledge Management (INAP), 2007.
2. CABEZA, D., HERMENEGILDO M., VARMA S., *The* PILLOW/CIAO *Library for Internet/WWW Programming using Computational Logic Systems*. Proc. of 1st Workshop on Logic Programming Tools for INTERNET Applications, JICSLP'96, 1996, Bonn, pp 72–90.
3. DENTI, E., OMICINI, A., RICCI, A.: TUPROLOG*: A Light-Weight Prolog for Internet Applications and Infrastructures*. Lecture Notes in Computer Science, Springer, 2001
4. HANUS, M.: *Type-Oriented Construction of Web User Interfaces*. Proc. 8th Intl. ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, (PPDP'06), 2006
5. JOHNSTON, B.: *Prolog Server Pages, A Web Programming Language*. http://www.benjaminjohnston.com.au/template.prolog?t=psp May 20, 2010
6. MANN Kito D.: *Javaserver Faces in Action*. Manning Publications Co., 2005.
7. ORACLE *Introducing Java Server Faces (JSF) to 4GL Developers*. Oracle, 2006.
8. PAULSON, L. D.: *Building Rich Web Applications with Ajax*. IEEE Computer, 38(10):1417, 2005.
9. PROTZENKO, J.: *XUL* Open Source Press, 2006
10. SCHNEIKER, C.; SEIPEL, D.; WEGSTEIN, W.; PRÄTOR, K.: *Declarative Parsing and Annotation of Electronic Dictionaries.* Proc. 6th Intl. Workshop on Natural Language Processing and Cognitive Science (NLPCS), 2009
11. SEHMI, A., KROENING, M.: WEBLS*: A custom* PROLOG *rule engine for providing web-based tech support*. Technical report, Amzi! inc.
12. SEIPEL, D.: *Processing XML-Documents in Prolog*. Proc. 17th Workshop on Logic Programming (WLP) 2002
13. SEIPEL, D.; PRÄTOR, K.: XML *Transformations Based on Logic Programming*. Proc. 18th Workshop on Logic Programming (WLP) 2005.
14. SUCIU, A., PUSZTAI, K., VANCEA, A.: *Prolog Server Pages, 2003*. Proc. of Roedunet Intl. Conference, 2003.
15. SUN MICROSYSTEMS, INC. *Mojarra Javaserver Faces - JSF 2.0 Datasheet* Sun Microsystems, 2009.
16. WIELEMAKER, J., HILDEBRAND, M., VAN OSSENBRUGGEN, J: *Prolog as the Fundament for Applications on the Semantic Web*, Proc. of the ICLP07 Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS), 2007.