Declarative Web Programming with PROLOG and XUL

Christian Schneiker and Dietmar Seipel

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

Abstract. Modern information systems are more often web-based than simple single PC desktop applications. In the last few years, developers have used common frameworks like GWT, JSF or similar to produce thin or rich client applications with the use of Java server technology for the backend part.
This paper introduces a new way of implementing thin clients with declarative web programming and PROLOG as a powerful server. The focus of the server lies in the integration of databases. GUI scaffolding on the basis of the defined data tables, database schema resolving for generic programming, and database triggered event handling make it possible to develop easy-to-read and reliable code.

1 Introduction

Before Tim Berners-Lee and fellows like Roy Fielding developed HTTP, HTML, and the first browser named WorldWideWeb, working with terminal sessions to connect to a mainframe were already a common scenario. High prices for single computers pushed the users to connect via a terminal, consisting of a monitor and a keyboard, to a single mainframe. These mainframes were high-performanced, and, of course, also very high-priced, but they gave many scientists the possibility to get *tuned in*.

Over the years, with the reduced hardware costs, nearly every single office and also homes have become equipped with a personal computer, not to mention mobile phones and other gadgets. The need for mainframes for computing subsided, and everyone used his own desktop version of the needed programs. With the introduction of the WWW in 1991 by Tim Berners-Lee, which started with a newsgroup message in *alt.hypertext*, the possibilities were already there for rich or thin client applications, but the existing slow bandwith made it impossible to fulfill the required user experience like speed, large amount of data, and high computer graphics.

In these days, internet bandwiths with 100 $MBit/s$ are becoming common, and multi-user applications with simultaniously connected users all over the world made it necessary to look for options. Based on HTTP, HTML, CSS and JavaScript and also other technologies, many client/server frameworks have been developed in the recent past. Java-based solutions have to be mentioned, like JSF [12] and GWT [3], as well as libraries for PROLOG, like PROLOG Server Faces (PSF) [8], *Type-Oriented Construction of Web User Interfaces* [4] or *An ER-based Framework for Declarative Web Programming* [5].

The aim of this paper is to introduce a PROLOG-based approach to declaratively design thin client applications with the XML dialect XUL (XML User Interface Language) and a few predefined JavaScript functions. Nearly all of the progam logic can rest on the server, which is also PROLOG-based, and therefore the whole power of PROLOG can be used. The client itself is OS independent and uses the standard look-and-feel of the operating system. Figure 1 shows a screenshot of such a client.

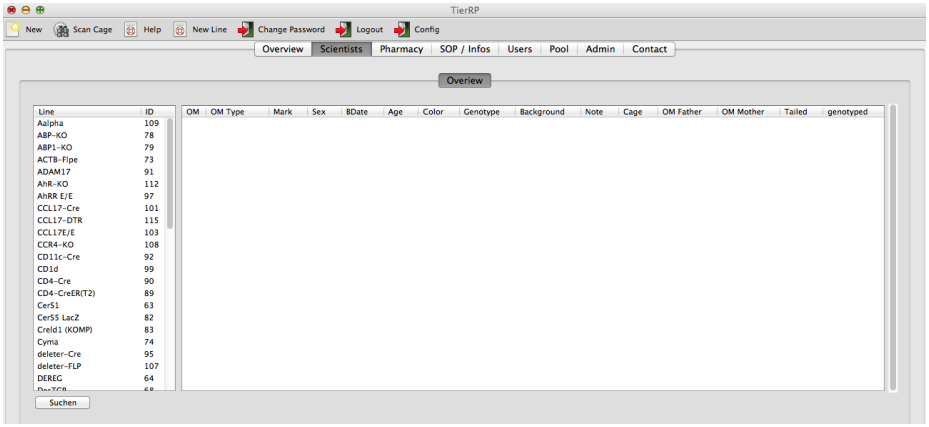

Figure 1. Screenshot of a XUL4PL based client application.

The main focus of our framework is the integration of databases. With XUL4PL it is possible to parse the involved database's schema for GUI scaffolding and generic code production. Thus, easy-to-read and highly reliable code could be developed. Another feature are database triggered GUI updates; on a multi-user system, all clients will be updated when a single user changes the data or even the structure of the database.

The rest of the paper is organized as follows: Section 2 gives an overview of XML-based programming technologies like JSF and PSF, XUL, and SOAP, as well as FNQUERY, a framework for efficiently processing XML data. The main components of XUL4PL will be described in Section 3. After a short overview of the implementation of the HTTP server, database programming with connection handling, GUI scaffolding and event triggers will be described in detail. Section 4 deals with the implementation of a client/server application.

2 XML Based Web Programming

This section describes the technolgies on which our framework XUL4PL is based. The framework itself relies on commonly used technologies, like JavaScript and HTTP connections. These techniques can be found in a wide varity of rich and thin client/server architectures. JSF and PSF are frameworks for Java and PROLOG for programming web applications, and they are using the same functionality in some parts. XUL is an XML-

based library for graphical user interface design, and SOAP handles the communication of data between the clients and the server. Both are extensivley used in our framework.

The following subsection describes the architectures on which XUL4PL relies as well as the used frameworks.

2.1 Web Applications

In the last few years, with the modernisation of the network infrastructure in both intranet and internet, it is common to develop new applications based on client/server technologies. The gain in bandwith gives the opportunity to handle large amounts of data by sending them over the net and to reduce the cost of high performance clients; the server is computing most of the program logic. While in rich client applications some of the logic still rests on the client, there are also a lot of thin clients, where nearly all of the data processing is handled by the server.

Rich client applications are a modern type of software clients which often include a unique framework for developing the client itself and also nested modules and plugins. It is common to give the user the opportunity to modify and extend the standard features. Depending on the used technology, the most used features of rich client applications are – with regard to above mentioned – OS independency, easy updateability, and the possibility of complex user interfaces; they can be used online and offline, because the whole logic is implemented on the client side and can be assisted by a server, if necessary. Data will be only transfered to the server for persistency and communication reasons. Widely used rich client applications are Eclipse and NetBeans for Java development, or Microsoft Visual Studio.

Thin client applications, on the other hand, only implement often used features of the backend logic, while most of the logic still rests on the server. Functions for GUI updates and small calculations are implemented on the client side. The advantages of thin client side applications are the huge scalability, OS independancy, and low costs for the hardware for the clients. A disadvantage is of course the need for working network connections. Common thin clients are, e.g., web browsers and terminal applications.

2.2 Common Web Frameworks

Implementing client applications from scratch is far away from state-of-the-art software development. Instead, frameworks are used for, e.g., GUI design, for handling the events sent by the server or the clients, and for the connection to any common database management system. Many different frameworks are brought to the developer; especially Oracle and Sun – with their programming language Java – have focused on the integration of their technologies with common HTML and JavaScript.

This section gives a short overview of Java Server Faces (JSF), implemented by Oracle and Sun, as well as of our previously developed client/server framework Prolog Server Faces (PSF). PSF also uses PROLOG for generating web applications, but with less functionality.

JavaServer Faces (JSF). As a framework for server-side user interface components, Sun Microsystems and other companies initially released JavaServer Faces in 2004, using 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 allows for developing server applications within the object-oriented programming language Java.

In JSF one can process client-generated events and alter states of components, making them event-oriented. It includes *backing beans*, which synchronize Java objects with UI 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 different ways, and it also offers many validation techniques.

Since it is a server-side technology, all pages requested by the client are preprocessed by the server. Via HTTP, every single requested XML document is transformed to standard XHTML, and nested calls to Java objects formulated 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, namely 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 for transforming 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 by adding them to the library.

Prolog Server Faces (PSF). PSF is a *stateful* and *event-driven* framework, that integrates logic programing in modern web applications. We are combining different techniques for mixing PROLOG with XHTML to develop *dynamic* web pages with the advantages of JSF for writing *condensed* XML. This XML will be expanded to 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 transforming XML elements, we extensively

use the XML transformation language FNTRANSFORM [10], which will be sketched in Subsection 2.5.

Like in JSF, nearly every XHTML element can be written in a compact form with additional attribute values, which read the data from complex term data 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 either read from an XML document or from a PROLOG data structure. The transformation itself is handled by FNTRANSFORM, which is integrated in our framework. When a client requests a PSF-XML file, the server automatically transforms it to XHTML with one of its request handlers.

2.3 The XML User Interface Language

The XML User Interface Language XUL is an XML dialect for declaratively defining graphical user interfaces. It has been developed by the Mozilla Foundation for implementing platform independent GUI's for their well-known browser Firefox and the email client Thunderbird. It is also used by a wide spectrum of companies for OS independent client/server applications like Google's AdWords tool. The following example shows a short code snippet of a modal dialog implemented in XUL.

```
<dialog id="newMessageDlg" ...>
   <script type="text/javascript" src="xulfunctions.js"/>
   <dialogheader title="Messages" description="new Messages"/>
   <vbox>
      <hbox>
         <label value="Priority:"/> <menulist> ... </menulist>
         <label value="Date:"/> <datepicker type="popup"/>
         <label value="Time:"/> <timepicker/>
      </hbox>
   </vbox>
</dialog>
```

All XUL elements could be combined with JavaScript and CSS, like in normal HTML webpages. Figure 2 shows the rendered dialog. XUL frames could either be rendered by using the runtime *xulrunner* – this is the most common way and applications like Firefox are startet like this – or by opening the XUL apps in Firefox.

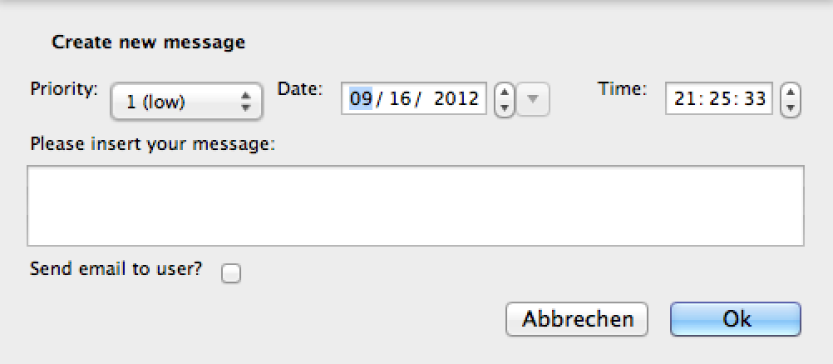

Figure 2. Screenshot of a XUL-based dialog.

2.4 Message Handling with SOAP

SOAP (Simple Object Access Protocol) is a network protocal relying on XML for passing information over a network between clients and a server.

The SOAP specification defines a messaging framework which normally consists of four parts:

- The processing model defining the rules for processing a SOAP message.
- The extensible model defining the concepts of features and modules.
- The underlying protocol binding for defining a binding to the underlying protocol like HTTP.
- The message construct defining the structure of a SOAP message.

The following example shows a SOAP message that is used for calling a function `GetStockPrice` – defined on the server side – with the parameter `IBM`. When the server has called the function, it can send back an answer in another SOAP message which the client can process, and based on which the client can, e.g., dynamically change the content of the GUI.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 299
SOAPAction: "http://www.w3.org/2003/05/soap-envelope"

<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header/>
   <soap:Body>
      <m:GetStockPrice xmlns:m="http://www.example.org/stock">
         <m:StockName>IBM</m:StockName>
      </m:GetStockPrice>
   </soap:Body>
</soap:Envelope>
```

2.5 FNQUERY and FNTRANSFORM

For the transformations in our framework, we extensively use the XML query, transformation and update language FNQUERY [10, 11], which is fully integrated in SWI PROLOG. Like in XPATH, it is possible to query complex structures with path expressions and axes. As an extension of XPATH, it is possible to select multiple branches over deeply nested structures. The sublanguage FNTRANSFORM, which extends XSLT, gives the user the feasibility to transform XML elements in PROLOG.

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 triples.

The path language FNPATH of FNQUERY is very similar to XPATH. Compound terms with the functor "/" are used for selecting subelements. The functor "@" is used for selecting the value of an attribute. 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].
```

3 The Framework XUL4PL for GUI Programming

For declarative GUI programming and native PROLOG rule implemantation, we have developed XUL4PL as a thin client framework. It is fully integrated into our SWI PROLOG framework, and it can be accessed online. The framework's HTTP server itself is implemented in PROLOG as well as functions for easy-to-use GUI and database handling. The user interface is based on XUL, for which we are using an extended version for communication and data exchange between client and server. We also use four JavaScript functions for calling the server and for sending and retrieving information. For defining the GUI, we use an extended version of XUL for which we have implemented features for data communication and easy-to-use GUI programming.

The following subsections describe our framework XUL4PL in detail with respect to the implementation of the HTTP server. After this, we give an overview of the database support, database driven GUI scaffolding and updates to the user interface with database triggers. An advanced method for message handling with SOAP is described afterwards.

3.1 HTTP Server Connection

The XUL4PL HTTP server is completely implemented in PROLOG using the HTTP support package of SWI PROLOG, implemented by Jan Wielemaker [13]. With the package, it is possible to handle data requests with GET and POST methods; even JSON data structures are possible.

In our framework, we have implemented four different handlers for processing data from the client or sending requested data, which are integrated into the DOM of the XUL-GUI. Our approach is dealing with POST data, which the server can process with the following handler:

```
handle(Request) :-
   member(method(post), Request),
   post_xml_to_fn_term(Request, FN),
   format('Content-type: text/xml; charset=utf-8~n~n'),
   format('<temp xmlns="http://www.mozilla.org/
      keymaster/gatekeeper/there.is.only.xul">'),
   apply_goal_from_fn(FN),
   format('</temp>').
```

Whenever a JavaScript function for the communication with the server is called on the client side, this handler parses the given information, i.e., the predicate to call with all the parameters like values from GUI input fields or constant values. With this, it is possible to call nearly all PROLOG rules from the client. The data from the client is sent in an XML message envelope containing the data for the predicate which the server has to call.

```
<message>
   <goal>predicate name</goal>
   <parameter>parameter 1</parameter>
   <parameter>parameter 2</parameter>
   ...
   <parameter>parameter n</parameter>
</message>
```

The predicate `apply_goal_from_fn/1` reads the message and retrieves the information for the goal, which it has to call, together with the additional parameters. The parameters can be different values from the elements of the XUL file or complete XUL documents, which can be processed themselves. We extensively use FNQUERY to parse the information given by the message.

3.2 Database Programming

A more efficient way of defining thin clients with XUL4PL is to connect them with a database. In this subsection, we describe how to connect our framework with a database, GUI scaffolding and the use of database triggers.

Database Statements. We have extended the list valid of XUL attributes to specify additional information needed like the database table name (`db_table`) and table column (`db_attribute`) where the data has to be stored or read;

When a GUI update is generated or the data are submitted to the server, then the XUL document structure will be parsed. E.g., to insert information into the database, for all XUL elements with the extended database attributes, we automatically call the predicate `xul_item_to_insert_or_update_statement/4`.

```
xul_item_to_insert_or_update_statements(
     Connection, Database, Xul, Statements) :-
   mysql_database_schema_to_xml(
      Connection, Database, DB_Schema),
   xul_form_to_inserts(Xul, Items),
   maplist(
      fn_item_to_insert_or_update_statement(
         Connection, DB_Schema, Database),
      Items, Statements ).
```

The predicate reads the used database schema from the data dictionary and checks the document with `xul_form_to_inserts/2` for the used attributes `db_database` and `db_table` and processes the data stored in the XUL file, e.g., input fields, radio buttons and drop down menues.

```
<hbox>
   <vbox>
      <label value="UserID" />
      <textbox width="80"
         db_table="User" db_attribute="User_ID" />
   </vbox>
   <vbox>
      <label value="First Name"/>
      <textbox db_table="User" db_attribute="First_Name"/>
   </vbox>
   <vbox>
      <label value="Last Name" />
      <textbox db_table="User" db_attribute="Last_Name"/>
   </vbox>
</hbox>
```

For XUL code above, the predicates parse the XML structure and generates SQL statements. It also automatically checks for the defined primary key and the assigned values.

```
xul_form_to_inserts(Xul, Item) :-
   D = descendant_or_self,
   findall( B:V,
      ( X := Xul/D::'*'::[@db_table=Table],
        Y := X/D::'*'::[@db_attribute=Attribute]/D::Tag,
        Tags = [listitem, menuitem, richlistitem, radio],
```

```
        ( member(Tag, Tags) ->
          true := Y@selected
        ; member(Tag, [checkbox]) ->
          true := Y@checked
        ; true ),
        xul_element_to_table_and_attribute(
           Y, Table->T, Attribute->A),
        V := Y@value,
        concat([T, ':', A], B) ),
      As ),
   xul_association_list_normalize(As, Bs),
   Item = row:Bs:[],
   !.
```

Afterwards, for all the processed data derived in the first step, the following predicate `fn_item_to_insert_or_update_statement/5` generates corresponding SQL statements. If a database row with the given value for the primary key already exists, then an update statement will be generated, otherwise an insert.

```
fn_item_to_insert_or_update_statement(
      Connection, DB_Schema, Database, Item, Statement) :-
   ( ( \+ fn_item_includes_primary_key(DB_Schema, Item)
     ; \+ fn_item_primary_key_is_in_database(
             Connection, DB_Schema, Database, Item) ) ->
     fn_item_to_insert_statement(
        Connection, Database, Item, Statement)
   ; fn_item_to_update_statement(
        Connection, DB_Schema, Database, Item, Statement) ).
```

Reading data from the database and generating XUL elements is also possible with XUL4PL and FNQUERY.

Database Driven GUI Scaffolding. An elegant way of defining user interfaces with XUL4PL is to use GUI scaffolding. We have implemented many predicates to automatically generate input elements based on the underlying database. In the following example, we will explain such a predicate, namely `odbc_attribute_to_fk_menulist/4`. Its arguments are the database, the table, and the attributes to be used. `Attribute` and `Table` are given as PROLOG terms representing XML structures, so-called FN-triples, because they can also be derived automatically for what we use XML in general.

```
odbc_attribute_to_fk_menulist(
      Connection, Database, Table, Attribute, Menus) :-
   A_Name := Attribute@name,
   T_Name := Table@name,
   [A_Name, Fk_Table] := Table/foreign_key-
      [/attribute@name, /references@table],
   mysql_use_database(Connection, Database),
```

```
concat(['SELECT ', A_Name, ' FROM ', Fk_Table], Statement),
odbc_query_to_tuples(Connection, Statement, Tuples_1),
sort(Tuples_1, Tuples_2),
( foreach([Tuple], Tuples_2), foreach(Item, Items) do
     Item = menuitem:[
        db_attribute:A_Name, db_table:T_Name,
        label:Tuple, value:Tuple ]:[] ),
Menus = row:[]:[
   label:[value:A_Name]:[],
   menulist:[]:[menupopup:[]:Items] ].
```

The predicate resolves the foreign keys given in `Table` and generates a menulist with all possible values. For these foreign key select menus, the referenced tables are read and only valid values are presented in the menu; the user cannot enter wrong data. In addition to the different foreign key values, it is also possible to include other values from the referenced table in the menu; the generated dialog is more readable.

The range of implemented predicates is huge: we have, for example, implemented the generation of trees used in applications like for file trees, complete input dialogs and wizards, as well as simple input types like listboxes, radio buttons and checkboxes.

Figure 3 shows an automatically generated XUL dialog. For the dialog, the database schema is read and drop-down menus with already assigned values corresponding to the foreign keys and input dialog are derived by our rule set.

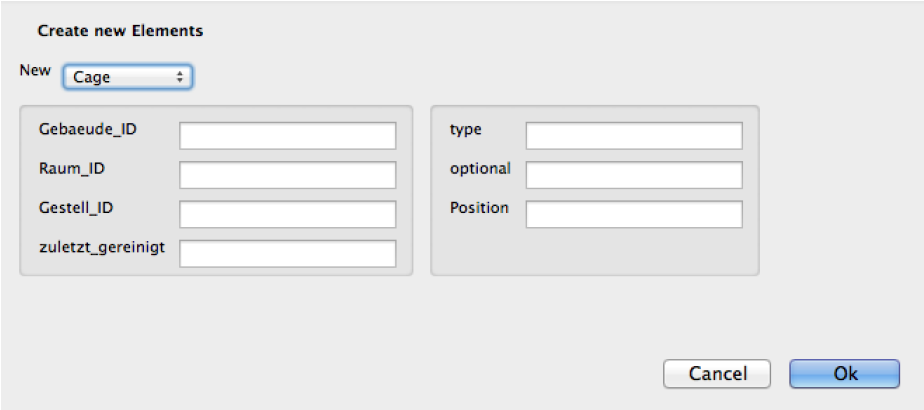

Figure 3. Automatically generated XUL dialog.

Database Triggered GUI Updates. Thin clients have the advantage, that they can be used in multi-user environments. When more than one user is working with such an application and data is transferred to the server, all the other apps have to recognize the change of data and have to be updated. Therefore, we use database triggers to react on such data changes.

When we insert into a database table, we call a predicate `xul_odbc_insert/3`, which generates the necessary insert or update statements and triggers events for the corresponding database table.

```
xul_odbc_insert(Connection, Database, Xul) :-
   xul_item_to_insert_or_update_statements(
      Connection, Database, Xul, Statements),
   ( foreach(Statement, Statements) do
        ( Statement = mysql_insert_tuple(
             Connection, EventDB:EventTable, _)
        ; Statement = mysql_update_table(
             Connection, EventDB:EventTable, _, _) ),
        call(Statement),
        ( xul_trigger_event(Connection, EventDB, EventTable),
          !
        ; writeln(user, noeventspec) ) ).
```

The `xul_trigger_event` predicate looks for registered listeners on all the acting databases. If a listener is registered with the call

```
xul_register_event_listener(+Type, +Goal, +Options)
```

then it calls the named `Goal` with `Options`, if available. `Type` is used for consistency like it is known from SQL: e.g., `ON UPDATE CASCADE` checks if nested GUI parts should also be updated.

```
xul_trigger_event(Connection, Database, Table) :-
  setof( Goal,
     ( xul_event_listener(Connection, Goal, Options),
       member(db(Database), Options),
       member(table(Table), Options) ),
     Goals ),
  ( foreach(Goal, Goals) do call(Goal) ).
```

3.3 Advanced Data Handling with SOAP

In the subsection above, we have described data handling with a short message envelope for sending and retrieving data. In modern client/server applications, it is common to use SOAP as a message format.

Therefore we have implemented a SOAP interface for the communication. The example below shows such a possible message format:

```
<soap:Envelope
   xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
   <soap:Header/>
   <soap:Body>
      <m:calculate_storage_price
```

```
            xmlns:m="http://www.uni-wuerzburg.de/xulpl">
            <m:ID>245</m:ID>
            <m:Date_From>2012-08-01</m:Date_From>
            <m:Date_Rage_Duration>5</m:Date_Rage_Duration>
        </m:calculate_storage_price>
    </soap:Body>
</soap:Envelope>
```

The message lets the server call `calculate_storage_price(2012-08-01, 5)`. The server itself can now process the data, and if necessary it could answer with another SOAP message.

4 Implementation of a Client/Server Application

For a case study on how to use XUL4PL and to test robustness and effectiveness, we have implemented a client/server application for a multi-user environment. The server itself is installed on a Ubuntu Linux machine, the clients are running under Windows, Linux and Mac OS X.

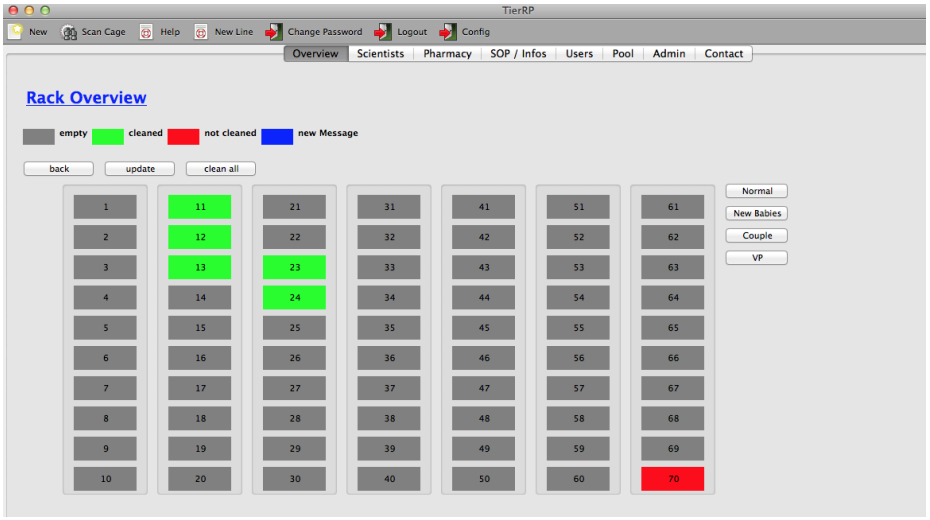

Figure 4. An interface implemented with XUL. The data is derived from a MYSQL database. The used OS is Mac OS Mountain Lion.

Figure 4 shows a part of a ressource planning system, which we have implemented with XUL4PL. The application consists of about 40 different dialogs, tabs, and windows – nearly all of them are generated automatically. The code for data retrieval could also be reduced to a minimum with our extended version of XUL. The code for the client/server application has about 2.800 lines of XUL code, the PROLOG code could be

reduced to only 3.000 lines with database techniques like GUI scaffolding. The lines of code of the framework itself is here excluded.

We have performed a stress test, where the database stored about 80.000 rows and 20 different users worked simultanously with the client.

5 Conclusions and Future Work

In this paper, we have introduced our framework XUL4PL for generating declarative thin client applications with a declarative GUI description language, and the combination of PROLOG as a backend-server.

With our framework, a developer has no need to implement new JavaScript functions but can focus on the definition of the PROLOG predicates for handling the application logic. The GUI can easily be defined with XML, and our extended version of XUL gives the user – in combination with our predicates for easily handling the design and behaviour – the opportunity to rapidly program user interfaces with the look-and-feel of the client's operating system. With the combination of XUL and ODBC, storage and retrieval of information with databases are easy to use and fast to implement. GUI scaffolding and database triggered event handling are one of our main features, which help to produce highly reliable code. Messages can be sent to both client and server with a short message format as well as with SOAP.

We have tested our framework with a ressource planning system, which is now used with a MYSQL and POSTGRESQL database and Linux as the server, the clients are currently running under Windows, Mac OS and Linux.

In the future, we will test the database connectivity with other databases like Oracle, DB2, and Microsoft SQL Server. Another feature will also be the integration of PROLOG Server Faces (PSF), such that we can easily switch between different kinds of user interfaces.

References

1. CABEZA, D., HERMENEGILDO M., VARMA S., *The* PILLOW/CIAO *Library for Internet/WWW Programming using Computational Logic Systems*. Proc. 1st Workshop on Logic Programming Tools for Internet Applications, JICSLP'96, 1996, Bonn, pp. 72–90.
2. DENTI, E., OMICINI, A., RICCI, A.: TUPROLOG*: A Light-Weight Prolog for Internet Applications and Infrastructures*. Proc. 3rd International Symposium on Practical Aspects of Declarative Languages (PADL), LNCS 1990, Springer, 2001.
3. GOOGLE, INC. *Google Web Toolkit - Developer's Guide*. Google, June 2012.
4. HANUS, M.: *Type-Oriented Construction of Web User Interfaces*. Proc. 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP), 2006.
5. HANUS, M.; KOSCHNICKE, S.: *An ER-based Framework for Declarative Web Programming*. Proc. 12th International Symposium on Practical Aspects of Declarative Languages (PADL), LNCS 5937, Springer, 2010.
6. PAULSON, L. D.: *Building Rich Web Applications with Ajax*. IEEE Computer, vol. 38 (10): 1417, 2005.
7. PROTZENKO, J.: *XUL* Open Source Press, 2006.

8. SCHNEIKER, C.; SEIPEL, D.; KHAMIS, M.: *Declarative Parsing and Annotation of Electronic Dictionaries.* Proc. 24th Workshop on Logic Programming (WLP), 2010.
9. SEHMI, A., KROENING, M.: WEBLS*: A Custom* PROLOG *Rule Engine for Providing Web-Based Tech Support*. Technical report, Amzi! inc.
10. SEIPEL, D.: *Processing XML-Documents in Prolog*. Proc. 17th Workshop on Logic Programming (WLP), 2002.
11. SEIPEL, D.; PRÄTOR, K.: XML *Transformations Based on Logic Programming*. Proc. 18th Workshop on Logic Programming (WLP), 2005.
12. SUN MICROSYSTEMS, INC. *Mojarra Javaserver Faces - JSF 2.0 Datasheet* Sun Microsystems, 2009.
13. WIELEMAKER, J., HILDEBRAND, M., VAN OSSENBRUGGEN, J: *Prolog as the Fundament for Applications on the Semantic Web*, Proc. ICLP Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS), 2007.