Overcoming the Multiplicity of Languages and Technologies for Web-based Development Using a Multi-paradigm Approach*

Sameh El-Ansary[1], Donatien Grolaux[2], Peter Van Roy[2] and Mahmoud Rafea[1]

[1] Swedish Institute of Computer Science, Sweden {sameh,mahmoud}@sics.se
[2] Universit Catholique de Louvain, Belgium {ned, pvr}@info.ucl.ac.be

Abstract. In this paper, we present QHTML, a library for building Web-based applications in Oz. QHTML provides the Oz programmer with a basic set of abstractions through which creating Web-based interfaces becomes similar to traditional graphical toolkits. In the mean time, QHTML is an experiment investigating whether a single language can replace the numerous ad-hoc combined languages/technologies currently used for building Web-based interfaces. QHTML is realized thanks to the multi-paradigm features of the Oz programming language, which supports symbolic data structures, a functional programming style, an object-oriented style and concurrency via dataflow and lightweight threads.

1 Introduction

Building Web-based applications requires the mastering of a number of languages/technologies (e.g. HTML, CSS, CGI, ASP, PHP, XML, etc..). Such languages and technologies were created to address different aspects on a by-need evolutionary manner. The result is a plethora of tools that are fitted together in an ad hoc fashion. Such languages/technologies include and are not limited to the following:

– HTML [1] is the main language for describing a Web-based Graphical User Interface (GUI) and it is a declarative language.
– A scripting language such as Javascript [2] or VBscript [3] for implementing any client-side dynamic behavior. Such scripts are imperative and are usually inlined inside the HTML code using a special tag to fit the declarative and the imperative flavors together.
– CGI Forms [4] provide a way to collect data from client-side. CGI Forms are neatly integrated in the syntax of HTML. However, server-side scripts written in ASP [5] or PHP [6] need to obtain the values entered by a user in the forms and repeatedly generate new pages to provide responses to client-side events. Server side-scripts in both ASP and PHP are also inlined with HTML using special tags.

*This work was funded at UCL and SICS by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under IST-2001-33234 PEPITO

– Cascading Style-Sheets (CSS) [7] are used for uniform and reusable formatting. CSS syntax is declarative, however, yet another syntax that needs to be learned.
– XML [8] and XSL [8] and the family of related standards are used for - among other functions - separation of content and presentation. This, despite its elegance, adds another burden of learning a new technology to achieve the separation aspect.

This multiplicity of languages and tools has negative implications on Web-based applications on the theoretical and practical levels. From a programming languages point of view, Web technologies lack a comprehensive model that accounts for all the aspects needed for a Web-based application. From the practical point of view, this multiplicity makes it harder to gain experience in Web development, as a developer needs to learn many technologies and languages and to know how to combine them together to achieve desired functionalities.

2 The QHTML approach

QHTML[3] provides an integrated model in the form of a set of abstractions to make it possible for the developer to build a Web-based application in the same way as any traditional application. In that model, the Web browser is perceived as a toolkit with well-defined geometry management capabilities.

QHTML is built as a layer over existing Web technologies such as Dynamic HTML (DHTML) [9] and the Document Object Model (DOM) [10].

QHTML is a module written in the multi-paradigm symbolic language Oz [11] and it is developed using the Mozart system [12]. Oz has support for declarative, functional, logic, concurrent, object-oriented and constraint programming. The diversity of paradigms in Oz has permitted the integration of many aspects involved in authoring Web-based applications in one language. The declarative paradigm in Oz is used to capture the declarative aspects such as interface description and style reuse. The functional and logic paradigms are used to provide the imperative aspects of interactivity, namely event-handling and dynamism of interface. The separation between the data and the presentation is enhanced by the advantages of symbolic manipulation of data to achieve several presentations for the same data by using different mapping functions. Communication between the client and the server is completely transparent to the application developer and is perceived as handling events in the graphical user interface. Abstractions for geometry management and more complete support for concurrency that accounts for threads is a consequence of having such an integrated model.

[3] A working implementation of QHTML could be obtained from: http://www.info.ucl.ac.be/people/ned/qhtml.

Fig. 1. A Basic Interface in QHTML

3 The QHTML Functionality

QHTML models the user interface as a structure of record values. An interface consists of a set of widgets, where each widget is specified by a record. Programming a complex interface then becomes a matter of doing computations on records. Since records are strongly supported by the Oz declarative pardigm, these calculations are efficient and easy to specify. This model is based on a similar model for higher abstraction for Tk [13] called QTK [14, 15], and it includes the following components: Windows, widgets, events, actions, handlers. In addition to those components, the model has support for enhanced semantics for geometry management, a more general concurrency support, and abstractions for context-sensitive interfaces.

3.1 Windows and Widgets

A window is a rectangular area of the screen that contains a set of widgets arranged hierarchically according to a particular layout. A widget is a GUI primitive that is represented visually on the screen and that contains an interaction protocol, which describes its interaction with a human user. An interaction protocol defines what information is displayed by the widget and what sequences of user commands and widget actions are acceptable. A widget is specified by a record, which gives its type and its initial state. The following is an example:

```
Descr=td(img(src:"mozart.gif")
         lr(
            button(value:"Hi")
            text(value:"Hello, please type here")))
Window={QHTML.build Descr}
{Window show}
```

In the above example, we have the `Descr` variable whose value is a record describing an interface where `img`, `lr` and `label` are widgets. The call `QHTML.build` creates a window containing that interface (figure 1) and the `show` method spawns the browser.

A stand-alone QHTML application uses the `show` method to spawn a browser process. A different method is used in case the interface is to be remotely loaded which is further discussed in section 4. The widgets `lr` and `td` are used to align

other widgets in horizontal or vertical fashions respectively and are covered in section 3.4 about geometry management.

The importance of having interface descriptions represented by Oz records lies in preserving the declarativeness of the GUI. Declarativeness simplifies the mapping data to GUI. Such mapping can be accomplished by coupling technologies like XML and XSL. The contribution of QHTML is that the data, the GUI and the mapping between them is all provided in one programming language under the same model. Style uniformity and reuse are also supported in the QHTML model by the `look` abstraction without any additional particular syntax, e.g. the following code creates a `look` that makes a button red and a label green and then applies this `look` to a GUI description.

```
MyLook={QHTML.newLook}
{MyLook.set button(backgroundColor:red)}
{MyLook.set label(backgroundColor:green)}
Descr=td(look:MyLook
         label(value:"Hello world")
         label(value:"Amazing colors !")
         button(value:"Ok")
         button(value:"Ok also")))
```

3.2 Handlers

A handler is an object with which the program can control a widget. Each widget can have a corresponding handler. Consider the following example:

```
Descr=td(text(value:"Hello, please type here" handle:T))
Window={QHTML.build Descr}
{Window show}
. . .
X = {T get()}
. . .
{T set("A new text")}
```

By having the handle `T`, we were able to get and set the contents of the text widget after the whole interface has been displayed. Observe that the variable `X` is a server side variable while `T` is referring to a text widget which is in the client/browser side. Nevertheless, this fact is completely transparent to the author of the Web-based application so he/she should no longer care about that difference.

3.3 Events and Actions

An event is a well-defined discrete interaction made by the external world on the user interface. An event is defined by its type, the time at which it occurs, and possibly some additional information (such as the mouse coordinates). Events are not seen directly by the program, but only indirectly by means of actions. An event can trigger the invocation of an action. An action is a procedure that

is invoked when a particular event occurs. Each widget has an associated list of actions. To illustrate actions, we can modify our simple example in the previous section as follows:

```
proc {Foo}
   X = {T get()}
end
Descr=td(text(value:"Hello, please type here" handle:T)
         button(value:"Hi" action:Foo))
```

Here we can see that we have exactly the same technique for handling events like HTML. Nevertheless, QHTML has the advantages of the integrated model because the actions, which are of an imperative nature, are written in the same language as the interface description, which is of a declarative nature. That was made possible by the functional paradigm in Oz where references to the action procedures are provided to the declarative description of the interface as values. The transparency property is also inherited from the model because when an author writes an action he/she does not differentiate whether this action is going to affect a server- or a client-side entity. This is a more intuitive and high-level way of performing interaction between the client and the server compared to the submission of a CGI form or the reloading of a certain PHP or ASP page. That transparency is a result of the integrated model implemented in one language.

3.4 Geometry Management

Current Web technologies lack a mature model for geometry management. It is the task of the HTML developer to figure out how to manipulate tables and frames to achieve a desired layout, which is a knowledge available only in the hands of experienced HTML programmers.

To cope with that, the integrated model also provides higher abstractions for geometry management. This is accomplished by having the top-down, left-right widgets: `td` and `lr` for controlling the placement of widgets and `tdframe` and `lrframe` for controlling relations between frames. That is in addition to a glue value that defines a constraint on the space taken by a widget and its response to resizing. The `td` and `lr` widgets are widget containers that pack widgets in a top-down and left-right orientation respectively. By default a widget takes only the space enough for it to be rendered and no more and is centered vertically and horizontally with respect to the containing widget/window. That layout could be further constrained by the glue feature that could be assigned the values `n`, `s`, `e` and `w` for north, south, east and west respectively or a combination of them. Gluing a side is asserting a constraint on that widget to let it always be tangent to its neighbor at that side and to occupy all the space available in the direction of that side. Gluing two opposite sides results in the widget taking all the space available in the direction of both these sides. This is illustrated by the following example:

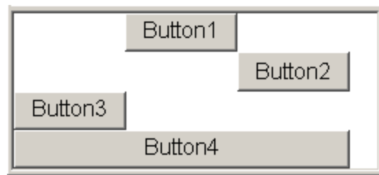

Fig. 2. Using Glue Values.

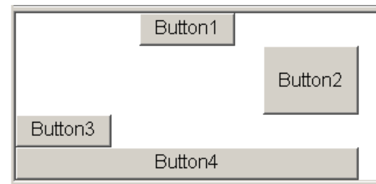

Fig. 3. Response of gluing to resizing.

```
td(glue:nswe
   button(value:"Button1")
   button(value:"Button2" glue:nse)
   button(value:"Button3" glue:w)
   button(value:"Button4" glue:we))
```

In figure 2, we see how the glue values affect the layout of the buttons. `Button1` is not glued, so it is centered in the containing widget `td`. The gluing of `Button2` and `Button3` to east and west respectively made them take the available space in the direction of the respective sides. The result of gluing to opposite sides is exemplified in `Button2` and `Button4`.

Figure 3 illustrates the response of the widgets to resizing. We see that the constraints are maintained and that `Button2` and `Button4` grew when they found available space.

It is also possible to have a grid structure where all widgets are organized in lines or columns of the same size (figure 4). The `lr` (resp. `td`) widget supports the `newline` special code which makes the following contained widgets jump to a new line (resp. column) right below the previous widgets, keeping the same column structure (resp. line) with the widgets above them. The following code exemplifies the use of `newline`:

```
lr(button(text:"One" glue:we) button(text:"Two" glue:we)
   button(text:"Three" glue:we)   newline
   button(text:"Four" glue:we) button(text:"Five" glue:we)
   button(text:"Six" glue:we)   newline
   button(text:"Seven"  glue:we) button(text:"Height" glue:we)
   button(text:"Nine"  glue:we)   newline
   empty button(text:"Zero" glue:we) continue)
```

The `empty` special code leaves an empty space in a line (resp. column) and the `continue` special code spans a widget over several columns (resp. lines). The same logic is extended for arranging frames using the `tdframe` and `lrframe` widgets.

3.5 Concurrency

The concurrency support in dynamic HTML documents is limited to a main event loop that serializes the events happening in the browser window in addition

Fig. 4. Using newline, empty and continue.

to an ad-hoc way to have more concurrency by inserting new events in the event queue after a certain time delay namely using Javascript timers. As a consequence of the integrated model, we can offer a more general support for concurrency by providing threads. In the Oz abstract machine, concurrency is supported by the ability to have many light-weight threads in the same operating system process [16]. The following is an example of how to integrate a thread easily in a Web-based application.

```
Window = {QHTML.build td(text(handle:E))}
{Window show}
. . . %%Some manipulations
thread
   {Wait X}
   {E set(value:X)}
end
. . . %%rest of maninputlations
```

Where `X` is some dataflow variable. In that example, we have created a thread that blocked on `X`, i.e, it waits until `X` is bound to a certain value and then executes its manipulation on the handle `E`. Again, notice that, `X` could be bound as a result of some GUI event or as a result of any other action in the application.

3.6 Dynamic Window Subparts

Another high-level abstraction offered by the QHTML model is the placeholder widget that is particularly useful for context-sensitive GUIs. The contents of a place-holder can change dynamically during GUI execution. A placeholder widget defines a rectangular area in the window that can contain any other widget(s) at any time as long as the window exists.

One can declare at any level of the GUI description a placeholder by writing `placeholder(handle:P)` where `P` is a handle to the placeholder. Afterwards, the application can set the contents of that placeholder to contain an arbitrary interface description and with the ability to change those contents later. Placeholders are particularly useful for building context-sensitive GUIs because one could specify a window with subparts that are sensitive to the available visual resources and who change their contents accordingly, e.g. an application can do the following: `{P set(label (text:"Hello") button(text:"World"))}` is filling the

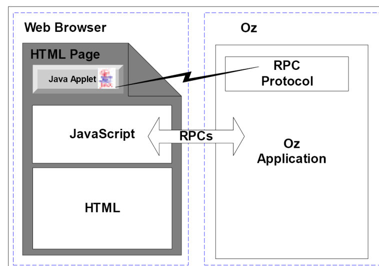

Fig. 5. Javascript-Oz RPC protocol architecture.

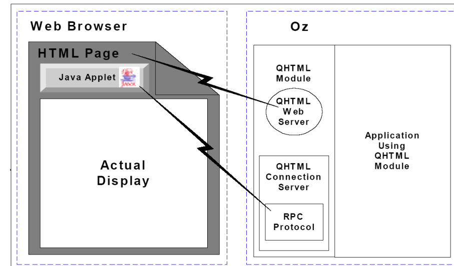

Fig. 6. Architecture of an application using QHTML.

placeholder with a label and a button after the detection of inadequate space, it can change the contents of the placeholder to: `{P set(button(text:"World"))}` where the placeholder is filled with a button only.

4 QHTML Architecture

The first requirement for the implementation of QHTML is a means of communication between the Oz language and a Web browser. To achieve that, we developed a simple Remote Procedure Call (RPC) protocol between Javascript and Oz, which is an independent component that could be used separately.

4.1 Javascript-Oz RPC Protocol

As illustrated in figure 5, the protocol has two components, an Oz component and a Javascript component. The Javascript component is primarily a Javascript interface to a Java applet, which is providing the real implementation of the protocol on the browser side. The Oz component is a module that could be embedded in any Oz application.

The technique makes use of the tight integration between Java and Javascript on the client-side. The Java applet provides socket communication with an Oz application. Consequently, any client-side Javascript function can use the Java applet to make RPCs to Oz procedures. Similarly, server-side Oz code can make RPCs to Javascript functions.

QHTML is a module that an Oz application links to (figure 6). Its main role is to hide all the details of the interaction with the browser. The steps for delivering the GUI and achieving that goal are as follows:

1. The Oz application builds the GUI description using record structures. For example something like: `{Descr=label(value:"Hello World")}`.
2. The Oz application requests the creation of the described GUI by executing the following statement. `{Window={QHTML.build Descr}}`. Consequently, the QHTML module transparently executes two operations:

(a) It creates a connection server, if it did not already exist, that waits for incoming connections. A connection represents an instance of the described GUI. `Window` is an object that maps to a particular connection in the connection server.
(b) It generates a page containing the RPC Java applet and an empty HTML display area, which we refer to by the connection page. Moreover, to facilitate the deployment of QHTML, a small HTTP server written in Oz was embedded inside the module in order to deliver the HTML connection page while its functionality is completely hidden from the application developer.

3. At that point, there are two ways to display the GUI: 1) A Web browser process is spawned locally by executing `{Window show}` where `show` is a method in the `Window` object. 2) A redirection page to the connection page is saved in a user-defined place by executing `{Window save("<Publicly - Accessible - Dir>")}`. In that way, it is accessible to other remote machines. In both cases, the connection page is generated with the right parameters to the Java applet depending on whether the browser is going to be spawned locally or will be used on a remote site.
4. Disregarding the Web browser spawning method, upon loading of the connection page, the Java applet runs and establishes a connection with the QHTML connection server. That connection becomes the communication link between the Web browser and the QHTML module and remains transparent to the Oz application.
5. Upon connection establishment, the QHTML module makes RPCs to generated Javascript functions that start to render the described interface using Dynamic HTML. After that GUI events can trigger server-side Oz code. Similarly, any events occurring on the Oz side can trigger GUI changes.

5 Conclusion

We have presented in this paper QHTML, a module for the Oz programming language that provides a model for symbolic authoring of Web-based applications through which the developer can treat the HTML document like a traditional graphical toolkit. QHTML makes use of: 1) The power of expression of the multi-paradigm language Oz. 2) An application architecture that realizes a transparent interaction mechanism between the Oz language and the Web browser.

The following table summarizes the aspects needed for the development of a Web-based application and enumerates the current languages/technologies that support those aspects in comparison with the Oz language constructs for supporting the same aspects in the QHTML approach.

In addition to the main advantage of the QHTML approach which is the integration of multiple aspects in one model implemented in a single language, two other features are realized:

1. A higher-level of abstraction for geometry management that simplifies the construction of complex layouts and that has support for context-sensitive interfaces.

Table 1. Summary of QHTML approach

Aspect	Current Techs.	QHTML Model
GUI description	HTML	Oz records
Events/Actions	Client-side scripts	Oz procedures & concurrency
Client/server communication	CGI/PHP/ASP	Oz-Javascript RPCs
Uniform styling	CSS	Oz records operations
Content-layout	XML + XSL	Oz records operations

2. A consequence of the integration of QHTML inside Oz is the ability to use threads which are the more intuitive and natural means of modeling concurrency. That is contrasted to the "hacky" incomplete way of using Javascript timers for supporting animation and similar tasks.

6 Related Work

We compare in this section our work to other approaches of authoring HTML-based GUIs through symbolic languages exemplified by Haskell [17] and Curry [18]:

1. Haskell, a purely functional language, has an HTML library [19] that aims at presenting Haskell data structures in HTML. For that, it encompasses some abstractions for building tables with constructs like "above", "below" and "beside". It also offers a mapping to all the HTML tags to make it easier for the Haskell programmer to generate HTML code. Thus, the Haskell HTML library is more of an interface rather than an abstraction layer in most of the parts except for the tables. There is no particular support for interaction other than by generating needed HTML tags.
2. Curry, a multi-paradigm language, provides an additional abstraction layer to HTML interfaces via its "High-Level Server Side Web Scripting" module [20]. In that module, syntactical details of HTML and passing of values with CGI are wrapped and abstracted as HTML forms. This leads to a high-level approach to server side programming that has the notions of event handlers, state variables and control of interaction. Despite of that, it does not account for any abstractions for better geometry management.

QHTML for Oz, shares with the libraries of Haskell and Curry the advantage of being able to symbolically transform data into user interface entities. QHTML shares the support for higher-level geometry management with the Haskell library but it has a more sophisticated model- based geometry abstractions that completely hide the notion of tables and is general for all interface elements. QHTML shares with the Curry library the support for interactivity and dynamism but without the administrative overhead for integrating programs with HTTP servers and with the ability to supply the user interface for Web sites or for stand-alone applications indifferently. Finally, QHTML has the advantage letting a developer treat the HTML document as a traditional graphical toolkit.

7 Limitations & Future Work

QHTML has some limitations and missing features where some of them constitute future work. Examples of those are the following:

1. In general, there is a performance penalty that is attributed to the fact that all events in the GUI are routed to the QHTML module first and are handled there, where communication is done over a stream connection between the Java applet at the browser side and the QHTML module at the server side. Nevertheless, The performance of a GUI constructed with QHTML is application-specific, in applications where the GUI contains many widgets, a QHTML implementation will perform better than a form-based implementation because changes will be done to individual widgets after an exchange of small messages with the server that is in contrast to reloading the whole page in a form-based approach.
2. QHTML has no support for vector graphics. The Scalable Vector Graphics (SVG) [21] recommendation is the most prominent candidate for integration within our model.
3. Due to the different implementations of standards in mainstream browsers, not all features could be provided in all browsers and some of them have to be implemented differently in different browsers. More work is planned to be done to achieve the same result on different Web browsers.

References

1. W3C: HTML 4.01 specification (1999) http://www.w3.org / TR / html4.
2. Netscape: Javascript documentation (2004) http://devedge.netscape.com.
3. Microsoft: VBScript documentation (2004) http:// msdn.microsoft.com/ library/ default.asp? url=/ library/ en-us/ script56/ html/ vbstutor.asp.
4. ncsa.uiuc.edu: The common gateway interface (2001) http:// hoohoo.ncsa.uiuc.edu/ cgi/ overview.html.
5. Microsoft: Active server pages (2004) http:// msdn.microsoft.com/ library/ default.asp? url=/ nhp/ default.asp? contentid = 28000522.
6. PHP: PHP homepage (2004) http://www.php.net.
7. W3C: Cascading style sheet level2, CSS2 specification (1998) http:// www.w3.org/ TR/ REC-CSS2.
8. W3C: The extensible markup language (2001) http://www.w3c.org/XML.
9. Goodman, D.: Dynamic HTML: The Definitve Reference. O'Reilly & Assoicates (1998)
10. W3C: Document object model (DOM) level 3 core specification (2001) http:// www.w3.org/ TR/ 2001/ WD-DOM-Level-3-Core-20010913/.
11. Smolka, G.: The Oz programming model. In van Leeuwen, J., ed.: Computer Science Today. Lecture Notes in Computer Science, vol. 1000. Springer-Verlag, Berlin (1995) 324–343
12. Consortium, M.: The mozart programming system homepage (2004) http://www.mozart-oz.org.
13. Ousterhout, J.: Tcl and the Tk Toolkit. Addison-Wesley (1994)

14. Grolaux, D., Roy, P.V.: `QTk` – an integrated model-based approach to designing executable user interfaces. In: 8. Lecture Notes in Computer Science, Glasgow, Scotland, Springer-Verlag (2001)
15. Grolaux, D., Roy, P.V., Vanderdonckt, J.: `QTk` – a mixed declarative/procedural approach for designing executable user interfaces. In: 8. Lecture Notes in Computer Science, Toronto, Canada, Springer-Verlag (2001)
16. Mehl, M.: The Oz Virtual Machine - Records, Transients, and Deep Guards. PhD thesis, Technische Fakultät der Universität des Saarlandes (1999)
17. Hudak, P.: The Haskell school of expression: learning functional programming through multimedia. Cambridge University Press (2000)
18. Hanus, M.: A unified computation model for functional and logic programming. In: Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97). (1997) 80–93
19. Gill, A.: The HTML library for haskell (1999) http:// www.cse.ogi.edu/ andy/ html/ intro.htm.
20. Hanus, M.: High-level server side Web scripting in curry. In: PADL'01. Volume 1990 of Lecture Notes in Computer Science. (2001) 76+
21. W3C: Scalable vector graphics (SVG) 1.0 specification (2001) http:// www.w3.org/ TR/ SVG.