Using XForms to Simplify Web Programming

Richard Cardone, Danny Soroker, Alpana Tiwari
IBM Watson Research Center
Hawthorne, NY 10532
{richcar, soroker, alpana} @ us.ibm.com

ABSTRACT

The difficulty of developing and deploying commercial web applications increases as the number of technologies they use increases and as the interactions between these technologies become more complex. This paper describes a way to avoid this increasing complexity by re-examining the basic requirements of web applications. Our approach is to first separate client concerns from server concerns, and then to reduce the interaction between client and server to its most elemental: parameter passing. We define a simplified programming model for form-based web applications and we use XForms and a subset of J2EE as enabling technologies. We describe our implementation of an MVC-based application builder for this model, which automatically generates the code needed to marshal input and output data between clients and servers. This marshalling uses type checking and other forms of validation on both clients and servers. We also show how our programming model and application builder support the customization of web applications for different execution targets, including, for example, different client devices.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – modules and interfaces, object-oriented design methods. D.2.3 [Software Engineering]: Coding Tools and Techniques – standards, object-oriented programming.

General Terms: Design, Standardization.

Keywords

Web application, visual builder, MVC, XForms, J2EE, Eclipse, XMLBeans.

1. INTRODUCTION

Using the terminology of a former era, one could describe web applications as client/server software that uses a universal client. This universal client is the web browser and the various standards that allow browsers to run on almost any computing device. In particular, the HTML standard is a cornerstone of the web. HTML 1.0 [5] provided basic document formatting and hyperlinks for online browsing; HTML 2.0 [6] ushered in a more dynamic, interactive web by defining forms to capture and submit user input.

This march towards dynamic web content has improved the web's utility and the experience of web users, but it has also led to more complexity in programming web applications. This complexity arises from three main sources. First, dynamic web pages are often generated on the fly, which makes application code harder to understand and makes troubleshooting more difficult because of the extra level of abstraction that one must consider. The pitfalls here are similar to those found with dynamically generated programs. Second, even when dynamic web pages exist as a single source code artifact, they are often a mixture of markup languages, client-side scripting code and server-side function calls, which makes them nearly unreadable. In addition, the skill set needed to comprehend such source code is continuously expanding, which again makes maintenance difficult. Third, the high number of software technologies used in some web applications makes those applications complicated to design and fragile to deploy and run. These technologies can include JavaScript [13], JavaServer Pages with taglibs [14], servlets [19], Struts [1], XSLT [28], DOM [28], SOAP [28], Web Services [28], Enterprise JavaBeans [18], Service Data Objects [18], etc., along with related protocols and configuration data. The complexity and overhead of combining these technologies can reduce performance and make runtime problems difficult to isolate.

The work described in this paper is an effort to get back to basics. We focus on *form-based web applications*, an important class of applications that solicit user input through form interfaces and then respond back to users with dynamic content. Specifically, we concentrate on the interaction between browsers and the server software with which they directly interact. We recognize three main requirements for this client/server interface: (1) to provide a responsive user experience, (2) to efficiently pass input and output data between browser and web server, and (3) to accommodate a diverse set of client platforms. Our goal is to meet these requirements while reducing the complexity of web application programming.

This paper makes the following contributions:

- We introduce a programming model that simplifies the design of form-based web applications by separating client-side XML markup from server-side programming language considerations.
- We show how to efficiently implement our programming model by using an MVC-based application builder and by automatically marshalling data between XML and Java.
- We provide a general-purpose approach to the *multi-targeting* of applications, in which multiple specializations of an application are developed in parallel.
- We describe our prototype application builder that tests our ideas and that includes visual editors, a code generator and a runtime library.

Our solution, *HopiXForms (HX)*, begins with the use of browsers that support an XForms processor. An XForms processor is essentially a virtual machine that interprets a well-

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

defined, declarative specification language for forms. The XForms markup language [11] minimizes the need for client-side scripting by allowing dynamic form data and events to be handled declaratively. This language is designed to be embedded in other XML languages, like XHTML. In fact, XForms 1.0 is an integral part of the XHTML 2.0 draft specification [2]. XForms processors are available for Internet Explorer, Mozilla, and for some PDA and mobile phone browsers [9][28].[1]

The use of XForms allows us to separate the data representation used on the client from the representation used on the server. In an XForms page, form data is represented as XML. On the server side, form data is represented in programming language structures native to the server program, which is convenient for server programmers. For example, form data is represented using JavaBeans on J2EE servers, which are the servers that we use in our implementation. A key feature of our approach is that page designers manage form data declaratively and server programmers manage form data imperatively, and neither group is required to manipulate the other group's representation. This separation is possible because our system automatically marshals form data between client and server representations.

Our solution also includes a Model-View-Controller (MVC) application builder that further separates concerns. Visual editors are used to specify an application's control flow separately from its page definitions. Our builder interprets the application control flow graph and generates the J2EE configuration that determines the transition between pages at runtime. In addition, since control flow and page definitions are separate, developers can use our development environment to customize both pages and application flow for different runtime environments.

This paper proceeds as follows. Section 2 provides the context for our approach with regard to other web technologies. Section 3 describes the design choices of our MVC application builder. Section 4 shows how we support the customization of an application for execution in different client environments. We then wrap up with related work and concluding remarks.

2. BACKGROUND

Viewed from end-to-end, web applications often have client, application server and backend server components, which suggests a three tiered architecture [8]. The Client Tier provides the user interface and consists of browsers that support standards like HTML, XML and HTTP. The Middle Tier communicates with the Client tier and executes business logic on an application server. The Backend Tier provides database support and connections to other enterprise-wide systems. Our work focuses on the Client Tier and the portion of the Middle Tier that communicates with the client. Two important considerations in this part of the architecture are how user interfaces are defined and how data are exchanged across the network.

2.1 Defining Interactive Forms

In form-based applications, the ability to collect user input and to respond with dynamic output is paramount. This demand for dynamic web content was first met using ad-hoc methods, such as CGI scripts that generate HTML response pages on the fly. The ascendancy of Java, along with performance and maintenance concerns about the CGI approach, led to the servlet programming model [14][19]. In this model, HTML response pages are generated by Java code running inside a well-defined container environment. Such servlet code typically combines strings of HTML markup under program control to generate HTML response pages. Unfortunately, this means that changes to web pages often require changes to Java code and, consequently, that page designers need to be familiar with Java.

JavaServer Pages (JSPs) invert the relationship between the host and embedded languages: In JSPs, Java code is embedded in HTML markup. This preserves the basic structure of an HTML document and allows Java calls to be placed only where dynamic content is needed. Conveniently, JSPs are transformed into servlets and then processed. On the other hand, JSPs continue to mix markup and Java code in the same source artifacts, which makes these artifacts difficult to understand especially when both server-executed Java code and client-executed scripting code are used in the same page.

The JSP lineage represents just one of the many approaches that have been used to increase the dynamic capabilities of web applications. For example, Microsoft's Active Server Pages (ASPs), like JSPs, mix client-executed markup and server-executed generative code to create dynamic web pages. Not surprising, this intermixing can make ASPs difficult to understand and maintain. To address the issue, ASP.NET [24] adds server controls, server events, and a server execution model to the processing of traditional client-side HTML, all of which increase the skill set needed by developers.

For all but the simplest web applications, web pages resemble compiled *object code* rather than human-readable *source code*, and one can argue that it is misguided to expect otherwise. Web pages provide the context for a diverse and constantly expanding set of technologies to interact, including audio, video, real-time graphics, interactive forms, and peer-to-peer networking. Both client-side scripting and server-side generative code allow web pages to deliver this highly dynamic and varied content. The argument is rather than thinking of web pages as source code, we should instead think of web pages a kind of object code that happens to use displayable characters. The goal, then, would be to build easy-to-use web programming tools that hide the complexity of the web pages that get deployed.

There are problems, however, with this idea of treating web pages as object code that can only be manipulated using high level programming tools. First, unless programming tools can quickly support the constantly evolving requirements of dynamic web applications, we will always be tempted to expose to developers the lower level client-side scripting and server-side generative code used in web pages. Unfortunately, it is difficult to provide even limited programming capabilities to developers without exposing them to the full complexity of these Turing-complete languages and their associated data models (e.g., client-side JavaScript and server-side Java). Second, the use of server-executed generative code makes it difficult to associate runtime browser errors with the server-side modules that generated the failing code. The obstacles here are that browser debug support is not standardized and that code generation adds another level of indirection between source and executable.

XForms [11], on the other hand, improves the comprehensibility of form-based web applications by defining a

[1] In addition, there are compilers that convert XHTML+XForms pages into HTML+JavaScript pages [16].

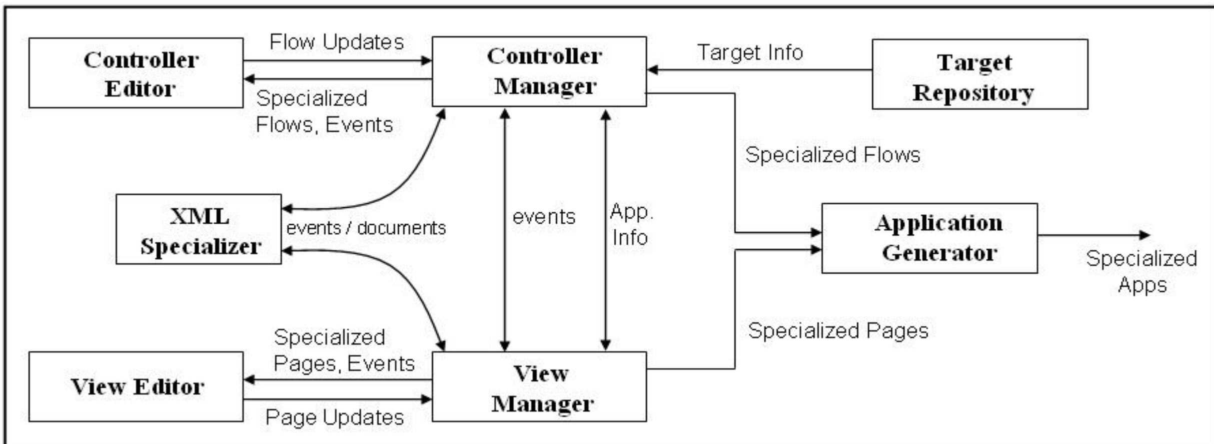

Figure 1 – HopiXForms (HX) Tool Architecture

consistent, declarative structure for dynamic forms. XForms is an embeddable XML language, which means that it is designed to be incorporated into other XML languages. In our case, we use XForms to extend XHTML markup with a more powerful forms capability. XForms also integrates existing markup technologies like XPath, XML Schema and Cascading Style Sheets [28]. XForms pages define form data, define UI controls, bind their controls to their data, interactively respond to events, validate input data, submit their data, and display dynamic content returned in responses. Most important, built-in event handling and validation functions remove the need for scripting in many XForms pages. Similarly, built-in support for dynamically changing a form's appearance removes the need for server-executed, generative code in most XForms pages.

To a page designer, XForms brings a rich, structured, declarative forms capability to existing markup languages. For the purposes of this paper, the details of the XForms markup language and the processors that implement it in browsers are not critical. We note that the W3C tracks the adoption of XForms on their web site [11] and that there are several freely available XForms implementations, including the formsPlayer™ [29] processor for Internet Explorer that we use. There are also several other good XForms references [10][23].

2.2 Exchanging Form Data

After form data are collected in a browser, they need to be submitted to a server. XForms submissions typically result in the transmission of a stream of XML data using an HTTP POST request. What data are sent, where the data are sent, and how the data are sent is all specified in the XForms markup.

On the server side, the ultimate representation for data is in the programming language structures in which the server program is written. In our application builder, this language is Java. To avoid exposing server programmers directly to the XML used on client, we generate JavaBeans from XML schemas during application development. At runtime, we automatically populate these JavaBeans with the incoming XML data and then pass those beans to the business logic code written by server programmers. After the business logic executes, the usual server response is to display dynamic content in an XForms page. Our runtime system automatically marshals data from output JavaBeans to XML instance data in the XForms response page. This XML instance data reflects the data model, not the desired appearance of the data in the view.

Wherever possible, our approach to building web applications is to separate concerns, to reduce the overall number of technologies required, and to reduce the skill set needed to develop the various components. On the client side, page designers create forms declaratively using XML languages. On the server side, programmers implement their business logic in Java without using XML technologies. Applications are developed by defining the control flow separately from application pages, which allows us to provide visual editors to support developers in each of these tasks.

3. BUILDING WEB APPLICATIONS

Our web application builder, *HopiXForms (HX)*, provides visual editors to define the controller and view portions of an MVC application. In addition, our editors and code generator use XML schemas to define the application's model. HX applications run in a J2EE servlet environment that supports Apache Struts [8]. Struts is a server-based facility that allows the control flow of a web application to be separated from the application's other concerns. The HX application builder, which we also refer to as the HX tool, generates the required Struts configuration to achieve this controller separation.

For illustrative purposes, we refer in this section to a sample application, called *SearchDate*, which calculates the day of the week for a given date. This application is adapted from sample code provided with the formsPlayer XForms processor.

3.1 HX Tool Architecture

The HX application builder is implemented as a set of plug-ins for the Eclipse platform [12], which provides a framework for integrating tool components. Figure 1 shows how the main components of HX's architecture reflect the MVC structure of HX applications. The Controller Editor is used to specify an application's control flow; the View Editor is used to specify an application's XHTML+XForms pages. As required by XForms, these pages also specify data models and their schemas. Schema can be created using existing Eclipse tools, such as an XML schema editor, that are not shown in the figure.

The Controller and View Editors each have a corresponding manager component that mediates between the editor and the rest

of the system. Editors receive data from their managers, such as events that affect the visual display of editable content. Editors also send data to their managers, such as update notifications that are of interest to other parts of the system.

One of the key ideas behind our system is that each web application is actually a family of application versions. This family consists of a common application and zero or more specialized versions that are tailored to specific execution *targets*. For each target, the controller and view portions of the application can be independently customized. For example, a specialized controller can disable control flow into part of an application that should not run in a certain target environment for security reasons. Similarly, a specialized view can be customized for screen size, screen resolution and other display characteristics of a particular device. Using the Controller and View Editors, developers can edit the artifacts of an existing specialization or create new specializations. These actions cause editors to interact with the Target Repository and the XML Specializer.

The Target Repository stores definitions of available execution targets, such as the set of available client devices or a set of end-user roles. Developers choose the targets for their applications through a UI mechanism associated with the Controller Editor, which serves as the focal point of application-wide information. When a developer selects a target, the Controller Editor displays the specialized flow for that target. HX control flows define the transitions between pages and are represented in XML; specialized flows are computed by the XML Specializer and communicated to the editor through the Controller Manager. View specialization works in essentially the same way, the main difference being that the XML view artifacts are XHTML+XForms pages.

The final architectural component is the Application Generator, which creates specialized web applications. For simplicity, we think of the common application as the base specialization. Thus, for each specialization target, the associated control flow and XHTML+XForms pages are taken as input to the generator and a deployable web application is outputted. These applications include the Struts artifacts generated from controller information and the JavaBeans generated from model information specified in views. The Application Generator supports incremental compilation, consistent with Eclipse platform philosophy.

The HX specialization subsystem consists of the Target Repository, the XML Specializer, specialization-aware editors, the code generator, and the specialization data and events that are exchanged between these components. The design of this subsystem extends that of IBM's Multi-Device Authoring Technology (MDAT) product [3]. We describe HX specialization in detail in Section 4.

During application development, HX interacts with other tools through the Eclipse infrastructure. For example, the Java Development Tools are used by developers to add custom Java code to skeleton classes created by the Application Generator. In addition, IBM enhancements to Eclipse [15] provide tools to create, validate, test and deploy XML artifacts and complete web applications.

3.2 Struts Overview

Before discussing how developers define the model, view and controller portions of HX applications, we provide background on how the Apache Struts [1][8] framework is used in HX. Architecturally, Struts is layered on top of J2EE servlet containers and is commonly installed on web servers such as Apache Tomcat [1]. During development, HX developers specify their application control flow graphically and HX automatically generates the necessary Struts artifacts that get deployed with the application. At runtime, these artifacts configure Struts to manage the transitions between application pages.

Struts delivers its controller function by providing a standard servlet class, which like all servlets executes on a web server in response to URI requests. The Struts servlet uses the configuration generated by HX before deployment to map application URIs to Struts *action classes*. In general, an action class is a developer-supplied class that the Struts framework invokes when a URI associated with that class is requested. During application development, HX generates action class skeletons to which developers can add their business logic. In Section 3.5, we describe how HX also generates JavaBean classes that hold web page I/O and how instances of these JavaBean classes are passed to action classes during request processing.

At runtime, application interaction typically proceeds as follows: A user enters form data on a web page and then presses a button, which causes a URI request to be submitted to the server along with the inputted data. On the server, the Struts servlet gets control and maps the requested URI to an action class. The action class is invoked and passed the input data. After the business logic in the action class executes, it returns a result called an *action-forward*. This action-forward is used by Struts to determine which response page should be sent to the user's browser.

3.3 Defining the Controller

The HX Controller models the high-level flow of client-server interaction within an application as a directed graph. This graph has two types of nodes and three types of edges (transitions). A *page node* represents an XForms page, which executes on the client. A *branch node* represents a decision point, which corresponds to a Struts action class that executes on the server. A *page-to-branch transition* models a request from a page, which corresponds to an XForms **submission** element. A *branch-to-page transition* models the outcome of the server-side action, which corresponds to a Struts action-forward that results in a response page being sent to the client. Similarly, a *branch-to-branch transition* models an action-forward that leads to a new action.

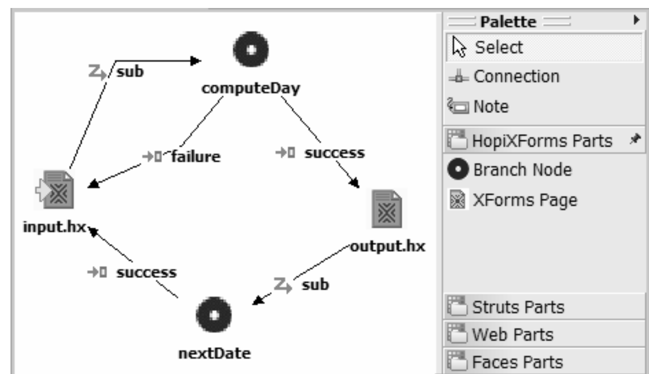

Figure 2 – Controller Editor

Figure 2 shows the control flow of our SearchDate sample application as visualized in the Controller Editor. The editor contains a palette for creating the different nodes and edges, and a

canvas area in which the flow graph is assembled. The application contains two pages – **input.hx** and **output.hx**. We use the *.hx* file extension to identify XHTML+XForms web pages. The arrow on the left side of the **input** page icon marks that page as the initial page of the application. At runtime, after entering the required data on the **input** page, a user would press a button to submit a request to the server. This submission, labeled **sub** near the top-left corner of the diagram, causes control and data to pass to an instance of the **computeDay** action class that is running on the server. When this Java code completes executing the application's business logic, it returns either **success,** to cause the **output** page to be returned to the client, or **failure**, to cause the **input** page to be re-displayed. The rest of the sample application's control flow operates in a similar way.

The Controller Editor encourages a top-down design of web applications. In addition to defining control flow, developers can navigate to other design artifacts from the editor. For example, double-clicking on a page node opens the corresponding page in the View Editor; double-clicking on a branch node opens the corresponding action class in the Java Editor. In addition, the Controller Editor reflects the *realization* state of the objects represented by the nodes and edges on the canvas. For example, if a node represents a page that does not exist in the file system, then that node is not realized and its icon is grayed out. When a page file is created it becomes realized and its corresponding icon is updated automatically. When an unrealized object is double-clicked, the Controller Editor orchestrates the creation of the node or edge by launching the appropriate wizard. Also, the editor decorates nodes and edges that have build problems with error markers.

This concept of realized and unrealized objects allows a loose synchronization between the Controller Editor and the concrete artifacts to which it refers. This approach simplifies implementation because even though changes in an application require the exchange of notifications between system components, the system does not have to guarantee consistency at all stages of development. Incomplete or inconsistent applications cannot be deployed, but inconsistencies in an application during development are presented as tasks yet to be completed. The Controller Editor is implemented as an extension of the Web Diagram Editor, which is part of IBM WebSphere Studio [15]. In addition, the Controller Editor works with standard Eclipse viewers, such as the outline viewer and the properties viewer.

3.4 Defining the View

The HX View Editor is used to create the XForms pages that define an application's view component. The language supported by the View Editor is XForms embedded in XHTML. XForms is designed as a modality-independent and device-independent XML language; the intent is that the same XForms document can be rendered on different devices using different interaction technologies, such as voice or stylus input. Thus, a WYSIWYG editor is not always the appropriate choice for displaying an XForms page. Even though our prototype focuses on the PC form factor using a standard web browser, we wanted to explore some of the presentation issues raised by the abstract nature of XForms. The View Editor uses a tree representation to reflect the hierarchical nature of XML-based web pages. Our editor includes wizards, context-sensitive menus, and drag-and-drop capabilities that make tree manipulation easier. When the View Editor is used in conjunction with a previewer, developers can see a concrete manifestation of their abstract pages. The editor also provides a read-only, source code view.

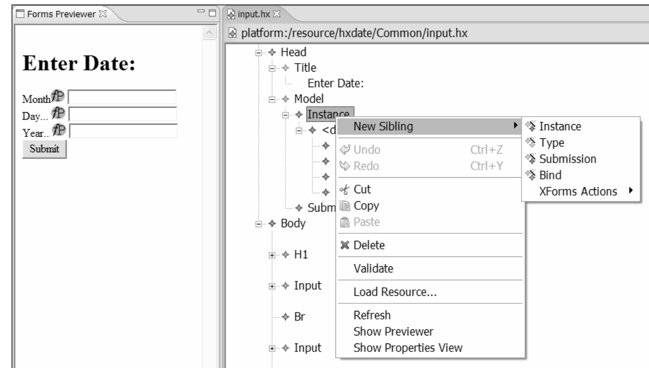

Figure 3 – View Editor

Figure 3 shows the input page of our SearchDate sample application in the View Editor. The previewer window on the left reflects the tree displayed in the edit window on the right. Updates to the tree are reflected in the previewer on saves. The previewer uses an XForms-enabled browser to display SearchDate's input page.

The edit window in Figure 3 also shows the context menu of the **instance** element. A key usability feature of our tree-based editor is the cascading context-sensitive menus that guide developers as they create page elements. These menus display the choices allowed by the XHTML+XForms schema when adding a child or sibling element to a page. The menus are tiered so that the most common choices are displayed most prominently.

The View Editor also defines wizards that assist developers as they construct their XForms pages. Creation wizards are invoked whenever an XForms element is added to a page. These wizards allow developers to conveniently assign the required or most commonly used attributes of the newly created element. For example, when an **input** control is created, the input wizard allows developers to associate the control with a model field in the page. In addition, the wizard creates a **label** child for the **input** control. In addition, edit wizards and the properties viewer can be used to change the attributes of existing elements.

3.5 Defining the Model

XForms pages typically define their data model using XML Schema [28]. HopiXForms provides wizards that help developers associate XML schemas with XForms **model** elements. During code generation, these schemas are used as input into an Apache XMLBeans [27] compiler, which generates the JavaBean classes that precisely represent in Java the data types specified in the schemas. These JavaBeans are not used on the client, but instead are packaged with deployed applications and used at runtime on the server to buffer XForms page I/O. Several other technologies, including JAXB [17], Castor [7] and EMF [12], generate JavaBeans from XML schemas; we chose Apache XMLBeans because it is mature, powerful and well-supported.

XForms processors running in browsers can automatically validate user input because of the strong typing provided by XML schemas. Similarly, the generated JavaBeans that execute on servers can provide the same level of type checking when bean values are assigned. This ability to check model data on both the

client and server requires no extra effort on the part of HX developers, but it promotes greater interactivity on the client interface and more robust server processing.

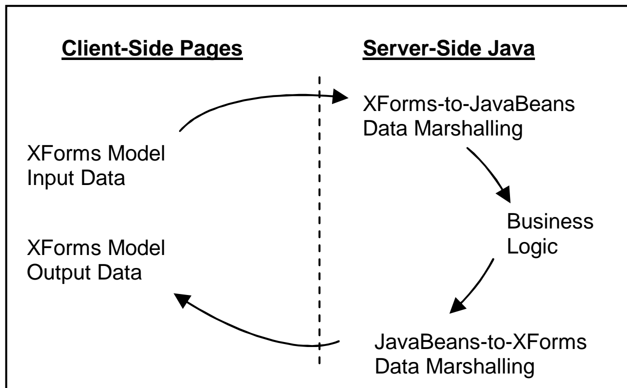

Figure 4 – Runtime Data Marshaling in HX

To complete our story, we need to describe how at runtime XML data on the client gets into JavaBeans on the server and vice versa. Figure 4 shows the flow of model data between the client and server. Starting on the client side, user input is captured by XForms as XML data. When the user initiates a submission, an HTTP POST request that contains the XML model data is sent to the server.[2] The HX runtime code on the server intercepts the incoming request and marshals the XML data into the appropriate JavaBean instance. The Struts framework then passes this bean to the developer-written business logic during Struts action class processing.

When the business logic executes, it can store response data in an output JavaBean, which can be a different instance or even a different type than the input JavaBean. When the business logic completes, Struts sends the appropriate response page to the client. The HX runtime code intercepts this outgoing response page and marshals the contents of the output JavaBean into an XForms **model** element. Finally, the HTTP response is sent to the client and the browser's XForms processor displays the dynamic content.

The automatic marshalling just described raises two interesting problems. First, the HX runtime system must determine the type of JavaBean into which the incoming XML request data should be marshaled. Second, the system must also determine precisely where in the XForms response page the outgoing XML response data should be placed. We now discuss our solution to these two problems.

Figure 5 shows the part of the input page of our SearchDate sample application that defines the XForms data model. The type definitions for the model are specified in the SearchDate.xsd schema, which is not shown but contains five string fields. These fields are also enumerated as grandchildren of the **instance** element. The **action** attribute of the **submission** element specifies the URL that is invoked to submit the **instance** data.

HopiXForms uses this model information to (1) generate the JavaBeans for the SearchDate.xsd schema and (2) indicate what JavaBean type should be used by the server for input data

```
<xforms:model schema="SearchDate.xsd">
 <xforms:instance>
   <data>
    <month/><day/><year/><dayOfWeek/><msg/>
   </data>
 </xforms:instance>
 <xforms:submission
   action="/hxdate/Common/computeDay.do"
   id="sub" method="post"/>
</xforms:model>
```

Figure 5 – Input Page Model

marshaling. The former task is handled using XMLBeans at application generation time as described above. The latter task can also be performed statically at generation time. One static approach is to configure a single input JavaBean type for the whole web application using initialization parameters in web.xml, which is the standard configuration file for web applications. A more flexible static approach configures each submission URL with its own input JavaBean type using parameters in struts-config.xml, which is the Struts configuration that specifies application control flow. HX implements both of these static approaches.

For even greater flexibility, however, the input JavaBean type can be determined dynamically when a request is received. A simple approach requires the addition of a parameter to the URL specified in the **action** attribute of the **submission** element. This submission-specific parameter specifies the input JavaBean type. HX automatically adds this parameter to the URL when the application is generated. This *mostly-dynamic* approach is efficient since each request specifies to the server the JavaBean type it requires and no further calculation is necessary. Most important, this approach requires no Java knowledge on the part of the page designer since HX automatically supplies the needed information. By default, HX uses this approach.

An alternative approach requires the server to inspect the root element of the incoming XML data and compare it to the root elements of the JavaBeans generated for the application. This completely dynamic approach requires unique root elements across an application's schemas. This approach, however, incurs some runtime overhead because it must inspect the XML payload and calculate the JavaBean type on each request. This approach is not currently implemented in HX.

The second interesting runtime problem involves marshaling data from JavaBeans into XForms response pages. Developers can specify an output JavaBean in the business logic code of their action classes or they can let the output bean default to be the same as the input bean. After an action class executes and just before a response page is sent to the client, the HX runtime system inspects the response page to determine which **instance** within which **model** element will receive the output data. By default, the first **instance** in the first **model** is chosen. Page designers can change this default behavior by adding an HX attribute to other **model** or **instance** elements in the page. This ability to add custom attributes is supported by XForms.

Once the target **instance** is known, HX determines if the **instance** is empty or if it has child elements. If the **instance** is empty, the JavaBean's contents are streamed into the **instance** in XML format. If the **instance** has children, then by performing a depth-first traversal starting at the **instance**'s root, HX matches

[2] HTTP GET requests are also supported.

the name of each child element with the corresponding field in the output JavaBean. Child elements of the **instance** that don't have matching JavaBean fields are not traversed. When the traversal reaches a leaf child that has a matching JavaBean field, the contents of that field are written to the child.

This basic name-matching algorithm is flexible in two ways. The first kind of flexibility is that different JavaBean types can populate the same XForms page. If tighter control is desired, then the output JavaBean type can be restricted to the type that corresponds to the target **model** element's schema. The second kind of flexibility is the way the algorithm supports XML Schema list types. When the traversal of an **instance** encounters the first XML element in a list, the list is replaced by all matching field elements in the JavaBean.

3.6 Developing HX Applications

Developers begin work on a new HX application by dragging and dropping page nodes and branch nodes onto the Controller Editor's canvas. The connections between these nodes define the control flow of the application. Whenever the controller information is saved, the HX incremental builder generates a skeleton Struts action class for each new branch node. The source code of these generated classes indicates where developers should insert their business logic code. The incremental builder also generates the Struts artifacts that will execute the application's control flow graph at runtime.

At any time, developers can edit the XForms pages that are represented by page nodes in the application's controller graph. XHTML+XForms markup is created using the View Editor. When a page is saved, the HX incremental builder generates the JavaBeans that correspond to the XML schemas specified in XForms **model** elements. These generated JavaBeans are used at runtime on the server to (1) pass request input data to action classes and (2) return response output data from action classes. HX never deletes code from action classes since they also contain developer-written code. Action classes, however, are automatically updated with new JavaBean types when new schemas are specified in XForms pages. In addition, HX adds the JavaBean type that should receive a **submission** element's request data to that element's **action** attribute.

When all nodes and edges in an application's control flow graph are realized, the application is ready to be deployed. The application can be deployed on any web server that supports J2EE web applications, version 2.3 or 2.4. An HX application contains all the web pages, JavaBeans, Struts actions classes, and configuration files needed to run. By default, HX applications also contain their own versions of the Struts 1.1 library and the HX runtime library. HX applications can be accessed from any XForms-enabled XHTML browser.

4. MULTI-TARGETING

HX supports the customization of applications for multiple execution targets, such as multiple devices or multiple end-user roles, through a mechanism we call *multi-targeting*. Each customized version of an application can add, remove or modify pages or control flow based on any criteria important to the developer. This notion of customization, called *specialization*, was first presented in the MDAT system and is further expanded in HX. In MDAT, developers specify a device-independent model, view and controller, and then customize the controller and some aspects of the view to create device-specific applications. These *specialized* applications require the translation of their device-independent views into device-specific markup.

In HX, however, no device-specific translation is needed since XForms-enabled browsers already tailor XForms pages to their host devices. The device-independence of XForms means that (1) HX does not need a view translation engine and (2) HX does not need detailed device profiles. As a matter of fact, targets like *Administrator*, *Guest* and *Motorola V710* are all handled the same way in HX: they are simply treated as identifiers. In HX, we can easily generalize the MDAT concept of *multi-device* applications to *multi-target* applications because HX does not need to understand target semantics. Of course, HX editors and previewers *could* use target-specific information to enhance the development experience, but this information is not architecturally necessary and not part of our prototype implementation.

The HX user interface for specialization is similar to the existing interface in MDAT.[3] A Target Editor associated with the Controller Editor lists the targets defined to the system by accessing a Target Repository. These targets can be structured hierarchically into categories and sub-categories. The Target Editor provides a way to add new targets to the repository and a way for developers to assign targets to their applications. Applications always have a base specialization, known as *common*, and zero or more other specialization targets.

When the Controller Editor opens, the common controller is displayed by default. The Controller Editor provides a drop-down list to select one of the other targets assigned to the application. When a new target is selected, the XML Specializer computes the new target's controller, which is then displayed in the editor. Subsequent editing operations apply to the new target's controller. The View Editor has a similar selection mechanism for specializing XForms pages.

HX specialization supports several key features. Most important, modifications to parent specializations are inherited by child specializations. Thus, any modification to the common version applies to all specializations. An important distinction between HX specialization and XML Schema subtyping is that HX specialization works on an individual document instance and XML subtyping works on the class of documents defined by a schema. Specialization can also change any aspect of view or controller content, as well as the parts of the model defined in the XForms pages. Also, since HX controller and view information is expressed in XML, the same specialization mechanism, the XML Specializer, can be used for both.

4.1 XML Specializer

The XML Specializer is responsible for constructing a specialization from a base document and a set of transformations. Figure 1 shows the Specializer communicating with the Controller and View Managers. The Specializer has two main functions:

1. For each specialization, maintain a set of deltas that describe how that specialization differs from its parent in the specialization hierarchy.

[3] HX specialization is not implemented.

2. For any given specialization *S*, compute its complete XML representation by applying in order all deltas along the path from the root (common) document to *S*.

The deltas capture the structural differences between two XML documents. Each delta corresponds to an atomic change, such as the insertion or removal of an XML element; the insertion, removal or modification of a value of an element; or the insertion, removal or modification of an element's attribute. The Specializer requires that all specializable elements be uniquely identified so that references to those elements remain valid as the document changes. Typically, any attribute that can appear on all (or almost all) elements of the document can be used to identify elements, as long as the attribute's values are unique within the document. We call this attribute the *id attribute*. Each delta refers to the element it modifies using that element's id attribute. An alternative design would be to use the element location (e.g., via an XPath expression), but element insertions and deletions become more difficult to process, and specialization becomes more fragile under this approach. The XML Specializer is a general purpose facility that can manage versions of any XML document that supports an id attribute.

The Controller and View Managers send edit messages and context information to the XML Specializer. When an editor opens, its manager notifies the Specializer that it is working on the common version of a document. When the developer switches to specialization *S*, the manager communicates that information. If the developer adds a new element *E* to the document, then the manager sends an edit message of the form (**add**, *E*, *pid*, *nsid*), which means "add element *E* as a child of the element with id *pid* such that *E*'s next sibling is the element with id *nsid*". The Specializer creates a corresponding delta entry in specialization *S*. Similarly, there are messages of the form (**remove**, *eid*), (**change**, *eid*, *string_value*) and (**change_attribute**, *eid*, *attr_name*, *attr_value*).

When the Specializer receives an edit message it may perform bookkeeping operations to guarantee the consistency of existing deltas. For example, if a previously created delta refers to a sibling element and the current edit operation removes that element, then the previously created delta needs to have its sibling reference updated.

The Specializer sends computed XML documents to the Controller and View Managers upon request. When a developer wants to edit specialization *S*, a manager sends a *specialize* request to the Specializer that includes the common document and the specialization name. The Specializer computes the XML document for *S* by applying in order the deltas along the path from the root document to *S*, and then sends the result back to the manager. The manager then passes the computed document to the editor.

4.2 Specializing the Controller and View

The Controller and View Editors are the developer-facing components through which specialization is defined and seen. These editors can reflect the effects of specialization by showing the differences between a specialization and its ancestors in the specialization hierarchy. These differences can be highlighted by using different colors or icons to distinguish inherited elements from elements added, removed or modified in the specialization. Additionally, the editors can toggle between showing and not showing the origins of elements in a specialization.

One of the UI challenges of specialization is to present differences along a complete path in the specialization hierarchy, from the specialization all the way up to the common document, in a way that avoids visual clutter and information overload. For instance, the Controller Editor is able to show differences between parent and child controller graphs in an understandable way because of the two dimensional nature of these graphs. The visual cues that differentiate parent nodes and edges from those of a child allow developers to clearly see how the two graphs differ. A scheme for overlaying parent and child graphs is implemented in the MDAT Controller Editor. In contrast, the View Editor is an abstract, tree-based editor, which reflects the *logical* structure of a page but not its actual presentation. The differences between parent and child pages in the View Editor are also at the logical level, which means that developers have to interpret the ultimate effect these differences will have in actual pages. One approach to making these differences more directly interpretable is to reflect them in the previewer.

Another difference between controller and view specialization is related to their underlying languages. The HX controller language describes a directed graph and the language is insensitive to element order. For example, nodes can be added or removed without regard to other nodes, and the same is true for edges. This order-insensitivity simplifies bookkeeping since remove operations do not require sibling references to be updated in existing deltas. This also means that the XML Specializer can be optimized for schemas in which order doesn't matter. On the other hand, the HX view language is sensitive to the order of elements since order affects page layout. Thus, the XML Specializer's job in this case is more involved.

5. RELATED WORK

HX builds upon the ideas of MDAT [3], which is an MVC-based application builder that creates web and portlet applications that run on multiple devices. In MDAT, developers create *generic* controller and view components that are device-independent. Whether these generic components run on the client or on the server, they all share a server-side JavaBeans model. In MDAT, view pages are defined as JavaServer Pages (JSPs) [14]. Specialization involves translating generic JSPs into JSPs that contain device-specific markup, such as WML or HTML. MDAT's view language, the language of its generic pages, is essentially a mixture of XHTML and XForms UI controls. On the other hand, HX's view language embeds the full, standards-compliant XForms language into XHTML. This XForms-centric approach uses an XML model in the view and leads to the separate client-server models described in this paper.

JavaServer Faces (JSF) [22] is a framework for Java-based web applications in which the application UI is constructed from reusable server-side components. These components are transformed into different concrete, client-side UIs through the use of *render kits*, which can accommodate both markup-based and API-based UIs. JSF provides a strongly typed event model that allows developers to write server-side handlers for events generated on clients. In addition, JSF specifies server-side model and controller components for web applications. By contrast, HX splits function more evenly between client and server. In HX, XForms processors running on clients provide the event handling, constraint checking and dynamic display capabilities needed for typical interactive user interfaces. HX also supports separate model representations for the client and server environments. JSF

is an emerging server-centric technology that relies on programming tools to insulate developers from the underlying complexities of the code it generates.

In his quest to teach web application programming, A. Lee notes that a chief reason why there are few advanced university courses on the subject is because of the "incredible range of different Web technologies each of which is constantly changing." [21] HX is one of many proposals that address this issue of complexity. In another proposal, Kojarski and Lorenz [20] identify two sources of complexity in web programming: *intra-crosscutting* is the tangling of application functionality, presentation and control concerns; and *inter-crosscutting* is the scattering of fragments of closely related code among application pages. To address these problems, the authors present WebJinn, an MVC-based tool for building web applications. Like HX, the MVC design pattern is used to alleviate intra-crosscutting. WebJinn, however, uses an aspect-oriented approach to address inter-crosscutting, while HX uses code generation to mitigate the problem of scattered model information. Another tool, the Wizard framework [26], also provides an MVC approach for fast prototyping form-based web applications. Unlike HX, this tool only generates the skeletons of web pages.

The use of XForms in web applications is gaining acceptance. Trewin, Zimmermann and Vanderheiden [25] compare XForms favorably to three other abstract UI specification languages in terms of applicability to any target, delivery context, personalization, extensibility and simplicity. Barton et al. [4] use XForms as a means to communicate data between web servers and sensor-enabled client devices, such as wireless digital cameras and PDAs. Form fields can be filled in directly with sensor data as well as with manually entered input. HX can be used to build such applications.

6. FUTURE WORK

In addition to completing our specialization subsystem and experimenting with several real-world applications, there are several avenues for future research. First, we would like to extend HX to support other modalities, such as voice or handwriting. XForms is modality-independent, but the technologies and architectures necessary to build non-graphical XForms applications are not well established. Second, we are exploring enhancements to our View Editor that include different ways of visualizing the model-view connections in a page; the tight integration of XPath, XML Schema and CSS editors; and support for defining XForms functions, events and actions. In addition, we would like to integrate a graphical page editor into HX to support layout editing on different form factors. Finally, we would like to enhance server-side validation by automatically running XForms constraint checking on the server when submissions are made. This means that both schema validation and XForms validation could run on both the client and server platforms without any extra work on the part of programmers. Client-side validation enhances the user experience; server-side validation makes applications more robust in the presences of ill-behaved clients. One implementation approach is to run an XForms processor on the server and correctly associated incoming XForms model data with the source page from which it originated.

7. CONCLUSION

The motivation for the work described in this paper is to reduce the complexity of form-based, web applications. Our approach is to separate concerns during web application development and to reduce the number of technologies that a developer must understand to create these applications. We decompose web applications into their constituent parts by placing view function on the client, controller function on the server, and model function on both client and server. These two model parts use representations that are natural for their respective environments and that are automatically synchronized by our runtime system. The net result is that our web pages are devoid of most of the client-side scripting and server-side generative code that we see in many applications. Typically, our web pages contain only declarative XHTML+XForms markup, which means that they avoid the complexity of mixing multiple programming languages with their different process models. This simplification is possible because XForms-enabled browsers provide the interactive and dynamic presentation capabilities required by today's web applications.

Our application builder provides visual editors that reflect the MVC structure of the web applications that we generate. This structure clearly defines the different skill sets needed to build an application: Web page developers need to understand declarative markup technology such as XHTML and XForms; business logic programmers need to understand Java and basic J2EE servlet programming. Instead of adding more software layers to the runtime stack, we try to recapture some of the simplicity of the early web by returning to declarative web pages and by solving the problem of marshalling dynamic content between client and server.

8. ACKNOWLEDGMENTS

We thank John Barton, Norman Cohen, Susan Spraragen and Bill Trautman for their invaluable comments and considerable effort in editing this paper. We also thank the anonymous reviewers for their insights and suggestions.

9. REFERENCES

[1] Apache Software Foundation. http://www.apache.org.

[2] Axelsson, J., Epperson, B., Ishikawa, M., McCarron, S., Navarro, A. and Pemberton, S. XHTML 2.0, *World-Wide Web Consortium*, (Working Draft 22) July 2004.

[3] Banavar, G. et al. An Authoring Technology for Multi-Device Web Applications, *IEEE Pervasive Computing*, vol. 3, no. 3, July/September 2004.

[4] Barton, J., Kindberg, T., Dai, H., Priyantha, N. and Al-bin-ali, F. Sensor-Enhanced Mobile Web Clients: An XForms Approach. *Proceedings of the Twelfth International Conference on World Wide Web* (WWW03), Budapest, 2003.

[5] Berners-Lee, T., and Connolly, D. Hypertext Markup Language (HTML), *Internet Engineering Task Force,* June 1993. http://www.w3.org/MarkUp/draft-ietf-iiir-html-01.txt

[6] Berners-Lee, T. and Connolly, D. Hypertext Markup Language – 2.0, RFC 1866, *Internet Engineering Task Force,* November 1995. http://www.ietf.org/rfc/rfc1866.txt

[7] Castor home page. http://www.castor.org

[8] Cavaness, C. *Programming Jakarta Struts*, O'Reilly and Associates, 2003.

[9] DataMovil by SATEC S.A., http://www.datamovil.info.

[10] Dubinko, M. *XForms Essentials*. O'Reilly and Associates, 2003.

[11] Dubinko, M., Klotz, L., Merrick, R. and Raman, T. XForms 1.0, *World-Wide Web Consortium*, (Recommendation) October 2003. http://www.w3.org/MarkUp/Forms.

[12] Eclipse tool platform. http://www.eclipse.org.

[13] ECMA International, Standard ECMA-262, *ECMAScript Language Specification*, 3rd edition, http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[14] Falkner, J. and Jones, K. *Servlets and JavaServer Pages.* Pearson Education, Inc., 2004.

[15] IBM Websphere Studio Application Developer. www.ibm.com.

[16] IBM XML Forms Package, April 9, 2003, *IBM Alphaworks*, http://www.alphaworks.ibm.com/tech/xmlforms.

[17] Java Architecture for XML Binding (JAXB). http://java.sun.com/xml/jaxb.

[18] Java Community Process site for Java standards such as Enterprise JavaBeans, Service Data Objects, etc., http://jcp.org.

[19] Java 2 Platform, Enterprise Edition (J2EE), http://java.sun.com/j2ee.

[20] Kojarski, S. and Lorenz, D. Domain Driven Web Development with WebJinn. *Companion of the 18th Annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA), Anaheim, California, 2003.

[21] Lee, A. A Manageable Web Software Architecture: Searching for Simplicity. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada, 2003.

[22] McClanahan, C., Burns, E. and Kitain, R. *JavaServer Faces Specification*, v1.1, rev. 01. Sun Microsystems, May 2004. http://java.sun.com/j2ee/javaserverfaces.

[23] Raman, T. V. *XForms, XML Powered Web Forms*. Addison-Wesley, 2004.

[24] Thai, T. and Lam, H. *.NET Framework Essentials*. O'Reilly and Associates, 2002.

[25] Trewin, S., Zimmermann, G. and Vanderheiden, G. Abstract User Interface Representations: How well do they support universal access? *Proceedings of the 2003 Conference on Universal Usability*, Vancouver, 2003.

[26] Turau, V. A Framework for Automatic Generation of Web-Based Data Entry Applications Based on XML. *Proceedings of the 2002 ACM Symposium on Applied Computing*, Madrid, 2002.

[27] XMLBeans home page. http://xmlbeans.apache.org

[28] World-Wide Web Consortium standards including XForms, XML Schema, XPath and Cascading Style Sheets. http://www.w3.org.

[29] x-port Ltd, home page for the formsPlayer XForms processor. http://www.formsplayer.com