Web Services Experience Language (WSXL)

IBM Note January 1 2002

Editors:

Angel Diaz, IBM (aldiaz@us.ibm.com)

John Lucassen, IBM (lucassen@us.ibm.com)

Charles F Wiecha, IBM (wiecha@us.ibm.com)

Authors:

Ali Arsanjani, IBM (arsanjan@us.ibm.com)

David Chamberlain, IBM (dchamberlain@us.ibm.com)

Dan Gisolfi, IBM (gisolfi@us.ibm.com)

Ravi Konuru, IBM (rkonuru@us.ibm.com)

Julie Macnaught, IBM (jmacna@us.ibm.com)

Stephane Maes, IBM (smaes@us.ibm.com)

Roland Merrick, IBM (roland_merrick@uk.ibm.com)

David Mundel, IBM (mundel@us.ibm.com)

TV Raman, IBM (tvraman@us.ibm.com)

Shankar Ramaswamy, IBM (shankar4@us.ibm.com)

Thomas Schaeck, IBM (schaeck@de.ibm.com)

Rich Thompson, IBM (richt2@us.ibm.com)

Abstract

WSXL (Web Services Experience Language) is a Web services centric component model for interactive Web applications, that is, for applications that provide a user experience across the Internet. WSXL is designed to achieve two main goals: enable businesses to deliver interactive Web applications through multiple distribution channels and enable new services or applications to be created by leveraging other interactive applications across the Web.

To accomplish these goals, all WSXL component services implement a set of base operations for life cycle management, accepting user input, and producing presentation markup. More sophisticated WSXL component services may be specialized to represent data, presentation, and control. WSXL also introduces a new description language to guide the adaptation of user experience to new distribution channels.

User experiences that are implemented using WSXL can be delivered to end users through a diversity of distribution channels

- for example, directly to a browser, indirectly through a portal, or by embedding into a third party interactive Web application. In addition, WSXL user experiences can easily be modified, adapted, aggregated, coordinated, synchronized or integrated, often by simple declarative means. New applications can be created by seamlessly combining WSXL applications and adapting them to new uses, to ultimately leverage a worldwide pallet of WSXL component services.

WSXL is built on widely accepted established and emerging open standards, and is designed to be independent of execution platform, browser, and presentation markup.

Status of This Document

Table of Contents

1.0 Introduction

1.1 Business Environment

In today's environment, businesses face an array of marketplace pressures. These pressures are often related to being able to offer a product or service in a timely fashion, exposing business processes to business partners, or offering access to legacy functionality in order to leverage existing investments. To be successful, businesses need to deploy applications that support their business processes quickly and economically, often relying on third parties in the process. They need to exploit an increasing diversity of on-line distribution models - including both direct and multiple distribution channels - in order to reach a variety of customers. When doing so, they may need to retain control of branding and user experience; with some distributors, they may choose to share some control, or give up control altogether. In some cases, they need to unbundle their business processes into core competencies that represent a set of more fundamental services. In this way, they can deliver a wide variety of increasingly sophisticated services and vary these services in response to changing market conditions.

The technical challenge common to all of these dynamic business requirements, is to be able to economically establish and maintain a variety of on-line distribution relationships in a way that can flexibly accommodate the demands of multiple business partners that use different business models. This challenge applies equally to producers of physical products and services and to producers of information or virtual products and services. The key differentiator in this arena is the ability to

rapidly (re-)configure the core competencies that drive the on-line distribution relationships.

In this environment, the core competencies of a business must be reusable for many different applications, including B2B and B2C. In addition, they must be configurable for different deployment scenarios, for example by branches, subsidiaries or business partners with different needs for process and branding.Finally, businesses often need the ability to create new integrated applications and services out of existing applications and services - either to create new product and service revenue streams or to reduce costs by eliminating redundancy and improving the information flow within the enterprise.

1.2 Requirements for a Web Services Experience Language

To be effective in this environment, businesses need a way to reduce the cost of developing interactive Web applications, delivering them to end users, and adjusting them to the changing requirements imposed by new business models.

This new approach must enable enterprises to:

1. Achieve an effective user experience that is appealing, efficient, and appropriately branded within each delivery channel
2. Support business flexibility through rapid reconfiguration, for example to add new distribution partners and application content providers quickly, or to change and update applications both collaboratively (jointly with partners) and autonomously (without being dependent on the partners)
3. Leverage existing skills and infrastructure

Moreover, the new approach needs to support:

1. Ease of change: to implement new and rapidly evolving business models, such as cross-selling, joint offerings, and outsourcing
2. Ease of adaptation: to deliver an application through new channels, such as a portals or third party Web sites, and through new business models, such as OEM or co-branding
3. Ease of aggregation: to group several applications into a single point of access, such as a portal
4. Ease of integration: to create a user experience by seamlessly combining several independently created applications
5. Interoperability: between applications and aggregation products such as portals, and between syndicated applications and distribution channels

1.3 Where WSXL Adds Value

Today, the business problems described in the previous section are addressed by a variety of different product concepts, architectures, and technologies, which typically focus on a particular problem. WSXL is intended as a technology that can improve, or add new capabilities to, a variety of various point solutions and their disparate application development and deployment mechanisms, while reducing the overall cost.

1.3.1 Interactive Web applications

Interactive Web applications are applications in which an end-user connects directly to a Web experience provider, and are the most familiar type of application on today's Web. WSXL enables modular construction of interactive Web applications by plugging together appropriate back ends, presentation designs, and control flows. WSXL leverages powerful design patterns such as XFORMS, combined with the rapidly emerging Web services infrastructure, to achieve ease of change, adaptation, and integration.

1.3.2 Application Syndication

Application syndication enables interactive Web applications to be delivered through different distribution channels. Application syndication typically involves the adaptation of the "look and feel" of an application to meet the branding requirements of the delivery channel. Application syndication typically result in "side by side" presentation, in which the presentation of one application is simply embedded in the presentation of another application.

This approach has the virtue of simplicity, but it does not allow the provider and the distributor to take full advantage of the ability to add value by combining their applications and integrating their business models. For example, the distributor's application is typically unaware of the page content that is delivered to the user, and unable to alter this content or its behavior. Moreover, an application that has been packaged for syndication is typically not interoperable with any portal,

which forces needless redevelopment of applications that need to be delivered through a variety of channels.

WSXL is designed to contribute to interoperability between syndicated applications and portals, thereby lowering the cost of implementing applications that support new business relationships. WSXL is also designed to facilitate the adaptation of an application to a particular delivery channel, thus enabling distributors to add value to the syndicated applications.

1.3.3 Portals

Portals enable the aggregation of interactive Web applications into a single point of access for end users and may provide additional functionality such as user management, application management, single sign-on, personalization, search, and so on. Each portal service or product typically has its own unique interface between applications and the portal, which limits interoperability.

WSXL is designed to contribute to interoperability by providing a technology that can form the basis for standardizing the interface between applications and portals. At the same time, WSXL is designed to provide a growth path for portals in terms of functionality. In particular, portals typically support "side by side" presentation: the applications that are aggregated are each assigned their own rectangular area on the display, and most portals are very limited in the ability of the aggregator to add value by creating a new user experience that tightly and seamlessly integrates the relevant portions of several existing user experiences. WSXL is designed to support a variety of user experiences ranging from simple "side by side" presentation to tightly integrated and interacting applications.

1.3.4 Service Providers and the Utility Model

Increasingly, user-facing business services are being implemented using a Service Provider or Utility-based model. The Service Provider model involves the delivery of software as a service to a large number of customers. In this model, it is necessary to be able to reconfigure the software services, including their presentations and packaging, in a variety of ways to meet the specific requirements for each customer. Moreover, it is essential that this reconfiguration can be performed not only by the service provider, but also by its channel partners, to enable distribution channel differentiation; and they must be able to charge for this reconfiguration as a service in itself.

The Utility-based model adds use-based pricing to the Service Provider model. It creates even greater demand for reconfiguration, by placing even greater pressure on the supplier to be responsive to diverse and changing customer wants and needs. The ability to customize the user or partner experience is a key element of the strategic value of the service provider and utility-based models, and must be accessible throughout the service delivery value chain.

With WSXL, service providers, e-utilities, and their distribution partners can rapidly compose new customer experiences, and wire them together to deliver a new experience that meets the requirements of a particular customer.

1.4 The Emerging Web Services stack

WSXL consists of a suite of web services, XML vocabularies, and user-facing processing models that build on a stack of existing and emerging Web services and XML standards. WSXL provides a Web services layer on top of the standards stack that is aimed at user-facing application development and deployment.

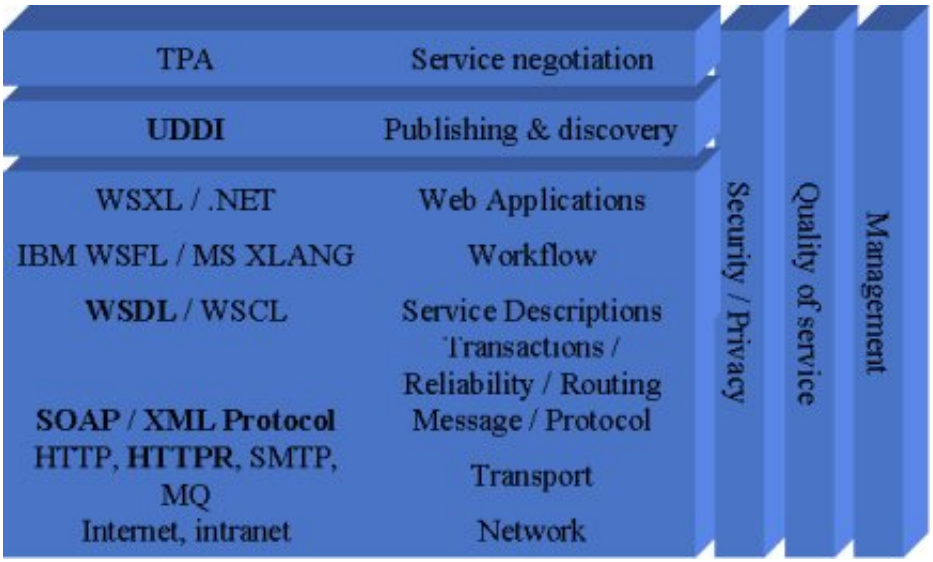

Figure 1: Web Services Related Standards

Figure 1 describes one view of the emerging web services standards stack. At the bottom layer TCP/IP has become the de facto network communication infrastructure. HTTP version 1.1 serves as the base upon which the IBM proposed HTTPR adds reliability. SOAP provides an XML based messaging protocol that can be deployed on variety of network layers such as HTTP and HTTPR. ' The Web Services Description Language (WSDL) proposes a way of describing software function. WSDL can use SOAP as the messaging protocol for invoking functions. WSFL and XLang describe how web services can be choreographed to model a business work flow. The three elements down the right-hand side have to be considered at all levels of the stack.

WSXL uses WSDL to describe interfaces specific to user-facing applications. WSXL application flow and logic is based on WSFL and XML Events.

As the world of Web services based solutions evolves, IBM expects this stack to continue to grow to meet the needs of the marketplace.

1.5 Web Services Experience Language By Example

In the following examples, we demonstrate how WSXL can be used to implement a number of practical business scenarios.

1.5.1 Syndicated Purchasing of Traveler Checks

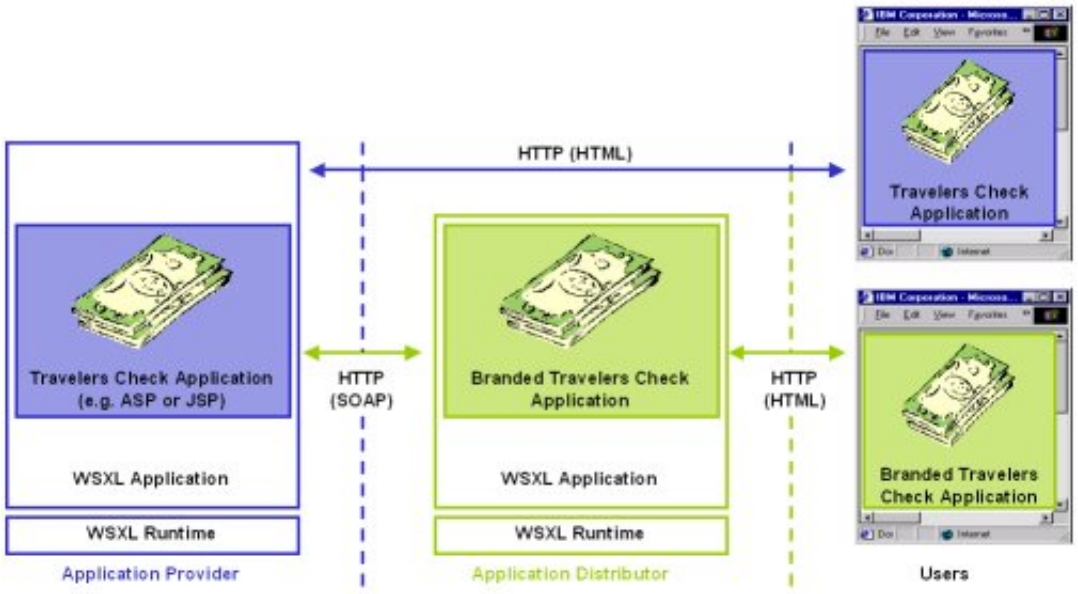

Figure 2: Syndicated Purchasing of Traveler Checks

In the case of syndication, WSXL gives providers the ability to use a single software asset across multiple distribution channels. As depicted in **Figure 2**, the provider can describe a WSXL application that wraps existing back end functionality. This application can then be adapted, if desired, and used to serve direct client access or embedded within a remote application redistributed by a business partner. In the case of the latter, the distributor would be able to establish a brand centric view of the application for its clientele by adapting the look, feel, content, flow, and functionality of the application as needed. Besides achieving application syndication with WSXL for presentation purposes, the provider's application can also have direct control over events raised by interactions at the browser. Moreover, reliance on widely adopted open specifications will eliminate the need to develop (or purchase) and maintain solutions that do not have as diverse of a usage model as WSXL applications.

1.5.2 Distributed Visualization Services (Remote Portal)

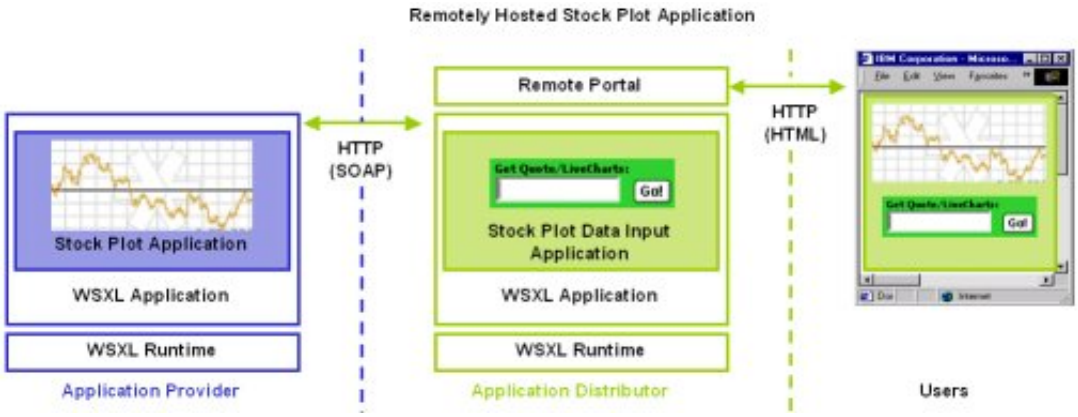

Figure 3: Distributed Visualization Services (Remote Portal)

Similarly, we have the scenario depicted in **Figure 3** whereby a supplier of financial graphics utility services enables the redistribution of these services. A portal company could in this case augment the utility service with a branded data input form. This approach would enable quick enhancements to the portfolio of portal services at a low cost of entry.

1.5.3 Distributed Visualization Services (Local Portal)

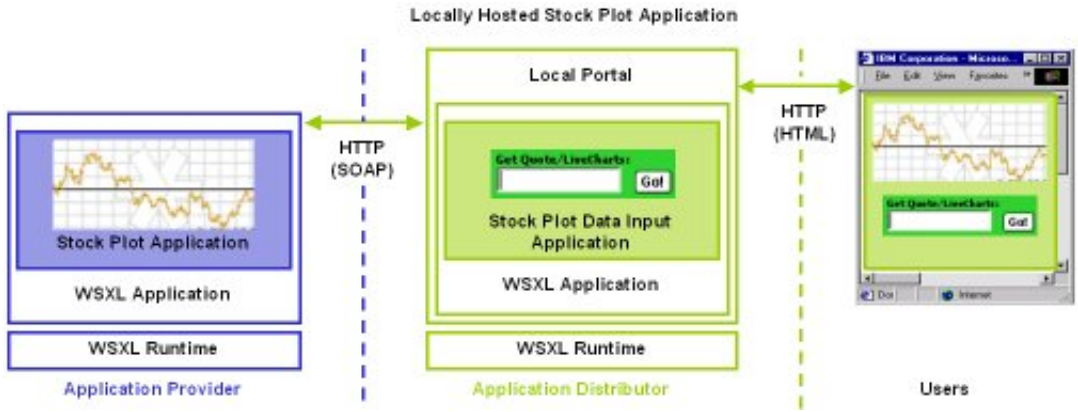

Figure 4: Distributed Visualization Services (Local Portal)

Building on the prior scenario, **Figure 4** demonstrates how a portal provider could also OEM a portal product that contains access to the financial graphics utilities. In this case, two levels of distributors in the application supply chain gain value from the same software service endpoint. First, we have the supplier of the portal product, who can be a value added reseller of the product that is installed in the brokerage firm. This supplier incorporates access to the financial graphics utility into its product portfolio of features. Through a revenue sharing agreement with the service provider, the portal product supplier can collect revenue on the sale of the product as well as collecting a percentage of each transaction. Second, the brokerage firm also benefits from the ability to brand remote services. Additionally, as with any ASP model the brokerage firm could also charge the client for access to these remote services. In either case, both the brokerage firm a nd the portal provider are able to address time to market needs by redistributing access to existing services.

1.5.4 Value Added Reselling of Traveler Checks

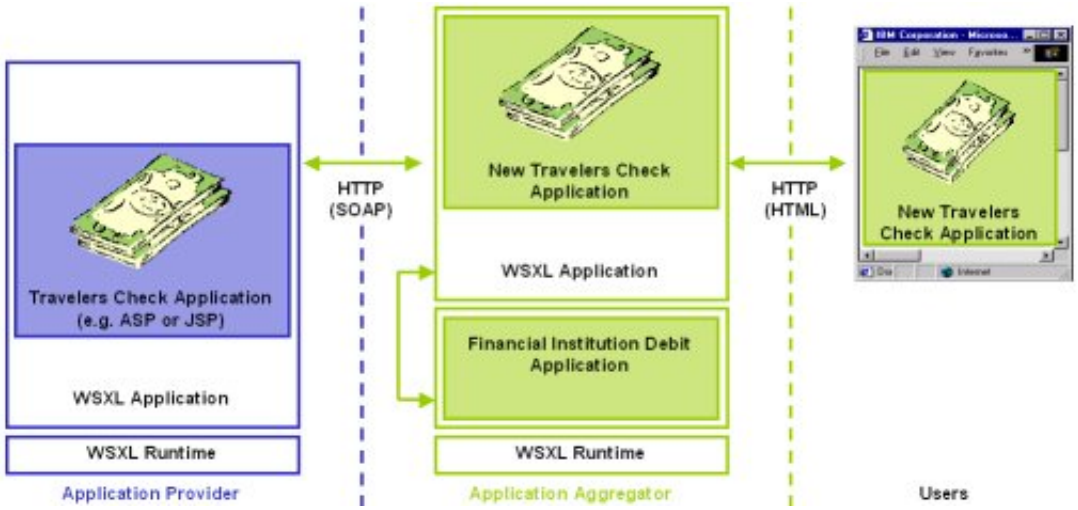

Figure 5:Value Added Reselling of Traveler Checks

Our last scenario, depicted by **Figure 5**, suggests how WSXL could be used to create new revenue generating applications. In this case the application domain represents the selling of Traveler Checks. However, the bank in this case is a value added reseller since it seeks to enhance existing application assets of their business partner (the Traveler Check Supplier) with a new payment instrument, specifically debit. In order to achieve this goal the bank needs to create a new application with a user experience that is unique to the bank. Using WSXL, the bank can achieve this by integrating the remote application of its business partner with a locally available application that addresses the handling of debit transactions from bank member accounts. In this scenario, the Traveler Check Supplier may, but need not, implement a new service operation that allows the purchase of the checks to be captured and settled out of band. WSXL offers a declarative language that allows collections of service interfaces to be grouped into reusable components that produce discrete units of work and can be used for business-to-business and application-to-application integration. WSXL is an enabling technology that makes it possible to achieve the business objectives in a timely and cost effective manner.

1.6 A Standards Initiative for Web Services Experience Integration

A standards initiative aimed at becoming the foundation for the next generation of interactive Web applications must address the needs of many constituencies, including end users, businesses, integrators, software developers, and platform vendors. Requirements from all of these constituencies demand a Web services experience integration model that:

1. Facilitates interoperability among disparate delivery channels. In particular, the approach must not impose a particular runtime model and must enable applications to be deployed easily for delivery through various channels.
2. Supports adaptation of Web applications. In particular, a Web application must be easy to "brand" and "co-brand" for the purposes of syndication.
3. Supports aggregation of Web applications into a larger user experience with no or minimal programming.
4. Provides for tight and fully customizable integration between Web applications, resulting into a seamless user experience.
5. Enables applications to be changed or modified easily. For example, business professionals must be able to quickly deploy and test new business models.

Moreover, these objectives must be met by reusing existing and de facto industry standards - in particular, XML based standards such as XPath, XML Events, DOM, XForms and XLink as well as Web Services standards such as SOAP, WSDL and WSFL. We believe that a combination of Web service based function and declarative specification of integration between such services will ensure broad acceptance across a wide range of constituencies.

2.0 Web Services Experience Language

The approach taken in WSXL to meet the requirements listed above is to allow a web application to export one or more component interfaces that expose enough information to enable adaptation, aggregation and integration while still allowing the applications to evolve. WSXL enables applications to be built out of separate presentation, data, and control components;

this helps developers to separate design issues that if left intermixed in more monolithic objects would make integration for re-use in multiple channels more difficult. To ensure a fit with existing web-based application architectures, WSXL services generate markup that can be used by conventional browsers and devices through existing formats and protocols. Targeting a wider market of users, channels and tasks requires a significantly greater number of variations than web applications currently support. WSXL reduces the cost of producing multiple variations of the same application by enabling the assembly of applications from a variety of components. This also allows applications and their components to start out simple and be refined in a business driven manner.

2.1 Introduction to the WSXL Framework

This section introduces the WSXL component framework for web applications (based on the well-known Model-View-Controller architecture) and relates it to existing standards such as XFORMS and XLINK. We also give a simple example of how data, presentation, and control components can be reused to produce new web applications by integration rather than programming them from scratch. The following sections describe the interfaces provided by data, presentation, control and their container, and give details of how the above scenarios could be implemented.

WSXL applications consist of one or more data and presentation components, together with a controller component which binds them together and specifies their interrelated behavior.

A WSXL base component has interfaces for life cycle management, event handling, and generation of output markup. The life cycle operations can be used to explicitly create and destroy instances of a WSXL base component. A WSXL base component may define XML events that it may raise, as well as XML events to which it can respond. A WSXL base component can generate output markup, in one or more target XML languages, upon request. Associated with a WSXL base component may be an Adaptation Description, which describes how the markup generated by the component may be adapted to new channels.

A WSXL data component is an extension of the base component that encapsulates a DOM-accessible representation of instance data and optionally associated model definitions. WSXL data components are based on the model and instance functionality of W3C XFORMS (see http://www.w3c.org/tr/xforms). Data components may be bound to presentation components using WSXL control components. In addition, data components may be connected to data sources external to WSXL applications, but this is an implementation detail that is beyond the scope of WSXL.

A WSXL presentation component is an extension of the base component that encapsulates a DOM-accessible representation of the elements in a user interface "page". The namespaces for elements used in presentation components are not fixed by WSXL, though commonly useful "widget" sets may be available such as those defined in the XFORMS UI draft. WSXL presentation components may be bound to data components using the WSXL control component and must raise and respond to events defined by that component for updating and resetting interaction data values and updating the presentation.

A WSXL control component is an extension of the base component that manages the micro-control between instantiated data and presentation components in order to bind them together. It, determines when to invoke event handlers to parse and reformat data values during updates in each direction. The WSXL control component reads declarations authored in an XLINK-based extensible control language specification to establish required arcs and implements the processing model for initiating, sequencing, and carrying out updates to synchronize the presentation with the state of the data. A second function of the control component is to manage the macro-control involving instantiation of entirely new data and presentation components as a user moves through the application. This aspect of control may be specified by state-machine notations or workflow languages using an extensible flow language interface.

The intent behind factoring a web application into separate data, presentation, and control components is to facilitate the reassembly of multiple alternative versions of the components in order to meet the requirements of separate channels, users, and tasks as outlined in the introductory requirements section above. As a result, WSXL will support the ability of web developers to assemble applications from data, presentation, and control components provided separately by independent vendors.

A key reason for factoring the overall design into a base component and extensions of that base component is to facilitate convergence between WSXL and other web services centric presentation component models.

WSXL interfaces can be described using the Web Services Description Language (WSDL). WSDL allows for a language-independent definition of the types, messages, and operations in a component's interface. WSDL separates these interface characteristics (the service's portType) from the binding of the portType to a particular transport mechanism. Many web services are accessed remotely over the network using SOAP messages. In other cases, components may be executed

locally and use higher performance "transport" such as direct method invocation. We intend the WSDL definitions in this document to be interpreted independently of whether the WSXL components described are to be accessed remotely or locally.

WXSL Adaptation Descriptions can be expressed using the Adaptation Description Language described in this document. We separate the Adaptation Description Language from the Web Services Description Language to allow a component interface to be associated with a choice of Adaptation Descriptions.

2.2 Component Descriptions

In this section we describe the interfaces provided by the base WSXL component. Next, we show how three particular types of components -- data, presentation, and control -- are derived from the base component interfaces. In the following section, we discuss how applications can be built from groups of components. The WSXL Container can be used to manage a set of components and nested containers. By further implementing the WSXL Component interface around the group it becomes reusable as a first-class component itself.

2.2.1 Base Component

Synopsis

The base component in WSXL specifies the interfaces that are common for all WSXL components: life cycle management, event handling, and generation of output markup. Other WSXL components, including data, presentation, and control, implement these base interfaces as well as the particular interfaces required for their functionality.

Base Component Interfaces

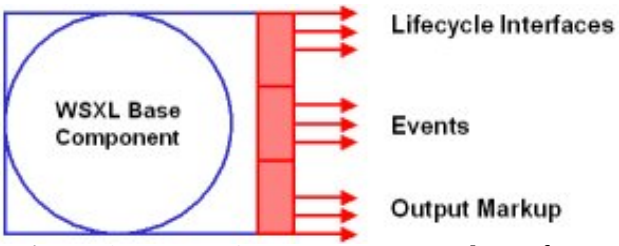

Figure 6: Base WSXL component and interfaces

The specific categories of interfaces of a WSXL Base Component are shown in **Figure 6** and include:

1. **Bindable**: Any WSXL object that supports clients establishing a WSDL binding to it implements the WSXLBindable interface. This uses the emerging concept of a WSDL Instance Binding to support runtime publishing of the binding portion of the WSDL definition. Implementation of this interface is required for all Component, Container and Event objects. It is also provides a standard means for other objects to indicate they support such bindings. The normative WSDL for this interface can be found in Appendix A.

   ### Interface: *WSXLBindable*

   getInterface() - This operation provides the means to request a WSDL Instance binding document.

2. **Lifecycle**: Both components and containers need to manage the Lifecycle of other objects and provide clients with means to indirectly refer to those objects in subsequent calls. We therefore introduce the concept of a 'Handle' as a remote reference to an object instance and define basic operations to create, initialize, destroy and get/set the properties of the object. These Lifecycle operations are defined in a shared interface ("WSXLBase") whose normative WSDL can be found in Appendix A. A null handle must be interpreted by compliant implementations as referring to the service itself.

Interface: *WSXLBase* (extends *WSXLBindable*)

createInstance(type, name, propertyList) - This operation provides a generic means of requesting the instantiation of a particular type of object. The supported set of object types should be defined as an enumerated set ("WSXLInstanceTypes") in the WSDL for the service. A Handle to the instantiated object is returned for use throughout the rest of the API. Containers (and others implementing the WSXLQueryable interface) require the name parameter be supplied. Other implementations should normally ignore this parameter. The propertyList parameter provides a set of initialization parameters to the object.

destroyInstance(handle) - This operation provides the means for explicit destruction of an object. For transitory objects this is often unnecessary though sometimes useful while for persistent objects it is often required. Subclasses will define the semantics of when an explicit destroyInstance invocation is required.

initInstance(handle, propertyList) - This operation provides the means to reinitialize the object the handle refers to. The semantics for this operation are to clear the current state and property information for the object and then initialize using the supplied propertyList. Components may also define semantics for use with persistent handles where this operation is used to initialize an instance of object referred to by the persistent handle for use during the current binding. The operation returns a handle for anytime that the service determines it needs to generate a new handle reference (eg. When the object, such as an HTTP session, the handle refers to has timed out) and therefore has been reset by the operation. This operation should be used for this reset purpose anytime another operation in the API returns an HandleNoLongerValid fault.

hasInterface(handle, portTypeName) - This operation provides the dynamic means for the client to determine if the object referred to by the handle supports a particular interface. The desired interface is indicated by the name of a portType and this operation returns a boolean indicating whether or not that portType is supported.

getInterface(handle) - Much as WSXLBindable provided the means for getting a WSDL Instance binding document for a service, this operation returns a WSDL Instance binding document for the object referenced by the handle. Null will be returned if the object does not support a separate binding to the instance.

invokeAction(handle, actionName, propertyList) - This operation provides the generic means of invoking the named action on the object referred to by the handle. The propertyList parameter provides any input the operation needs.

3. **Property management**: A WSXL component must implement property management operations whereby clients can modify properties at times other than initialization.

Interface: *WSXLProperties*

setProperties(handle, propertyList) - This provides the general means to set the properties of an object. It may return a PropertyList describing the properties that were modified (either directly or as a side effect of setting another property).

getProperties(handle, nameList) - General means for getting the properties of an object. If no nameList is supplied, all properties are enumerated. The returned document is a PropertyList.

4. **Events**: A WSXL component must implement event operations similar to DOM Level 2 Events. DOM Level 2 Events separate the concept of objects that may be the 'target' of an event from objects that may be an 'observer' for an event. Targets are conceptually where events happen while observers are propagation points where event notifications occur. Said another way, targets are where events are dispatched while observers are where event listeners may be registered. DOM Level 2 also defines a particular propagation mechanism for notifying registered listeners. While WSXL components are not required to implement the DOM Level 2 Event propagation mechanism, the separation of targets and observers has been maintained such that implementations may define a propagation mechanism of their own.

   This interface both allows clients to dispatch events the component has defined in its WSDL description and attach listeners interested in notification when such an event is dispatched. Means are also provided (in a separate interface) for handling events generated by other components such that an instance of a Control Component may cause this

operation to be triggered at appropriate times. The normative WSDL for these interfaces can be found in Appendix A.

Interface: *WSXLEventSource*

createEvent(eventInstanceType, propertyList) - Explicit means for creating an event object. Specifying a null eventInstanceType will result in the default event object being instantiated. A propertyList may be specified to set the properties of the newly instantiated event object. Note that while WSXLEvent objects are available for binding to separately, they are also accessible through their handles for most purposes. This reduces the complexity of code for the majority of users.

dispatchEvent(target, event) - This operation causes the referenced target object to 'dispatch' the referenced event object. The propagation of the event through the set of relevant observers (determined by the component's propagation mechanism) results in the notification of listeners.

queryAvailableEventTypes(observer) - This operation enables a client to dynamically query the set of event types a component internally could propagate to the specified observer. In addition to this dynamic means, WSXLComponent authors are encouraged to statically declare all eventTypes possible in the service's WSDL description.

addEventListener(observer, eventType, listener) - This operation permits clients to register for notifications of specific eventTypes that a component can propagate to the listeners registered at the object referenced by the observer handle. Specifying an eventType of null requests that the listener be notified of all events.

removeEventListener(observer, eventType, listener) - Clients should always remove events listeners when they are no longer interested in event notifications.

Interface: *WSXLEventSink*

handleEvent(listener, observer, target, eventType, event) - This operation is invoked on listeners when an event is propagated. A WSXLComponent must implement this in order for a Control Component to properly connect events between components. Note that most of the handles are references to objects on the Event Source, but that the listener handle is the indirect reference that was passed to the Event Source in an addEventListener() operation. Three commonly used fields in the processing of events have been called out as parameters in order to reduce the frequency of listeners establishing a separate binding to the event object.

5. **Markup**: Single operation providing the means for a client to request an object's markup. Returns an XML representation of the object's output. The normative WSDL for this interface can be found in Appendix A.

Interface: *WSXLMarkup*

getTemplate(handle, propertyList) - This operation returns a template (from an extensible set including for example document, schema, etc) for the markup the object referred to by the handle can generate. The propertyList parameter provides any input the operation needs.

getMarkup(handle, propertyList) - This operation provides a document fragment corresponding to the output of the referenced object. Passing a null handle will return the markup for the Component. A propertyList may be supplied for the objects use in the generation of the markup. An example of such a usage would be supplying a property indicating a delta from previous markup is preferred. The object may respect or ignore this request and if returning a document fragment delta, the fragment should indicate this in a direct fashion.

In general, a component may expose only a subset of its design externally -- by using an interface specification that reflects a restricted component structure and behavior intended for controlled extension or integration rather than the full underlying

implementation.

2.2.2 Data Component

Synopsis

In addition to the WSXL base component interfaces described above, the WSXL data component implements interfaces used to describe and maintain instances of DOM-accessible data in WSXL applications. The WSXL data component is based on the data functionality in XFORMS, and includes both XFORMS model and instance functions.

Data Component Interfaces

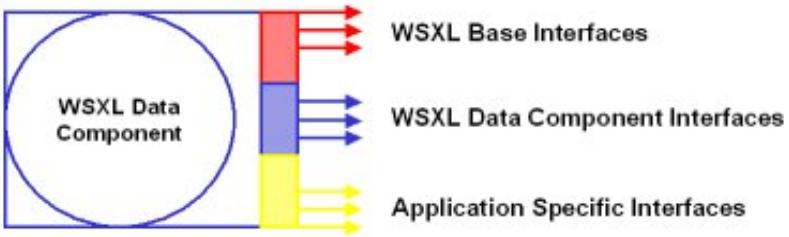

Figure 7: WSXL data component and interfaces

The additional categories of interfaces of a WSXL data component are shown in **Figure 7** and include:

1. **Access**: (Required) A WSXL data component describes the structure it wishes to expose using a flattened and subsetted version of the DOM Level 2 interface.

Interface: *WSXLDOMLite*

selectNodes(xpath) - This operation returns a HandleList of nodes that match the supplied xpath.

getNodeName(nodeHandle) - This operation returns the 'name' for this node. In components mapping other structures to the DOM API, this operation should return an indication of the field's name.

getNodeType(nodeHandle) - This operation returns the type of this node as a short as defined in the DOM API. Examples of types include Element, Attribute, Text, etc.

getNodeValue(nodeHandle) - This operation returns the 'value' of the node as defined in the DOM API. For components mapping other structures to the DOM API, this operation should return the field's value when possible.

setNodeValue(nodeHandle, nodeValue) - This operation sets the value that would then be available from the getNodeValue() operation.

2. **Access**: (Optional) A WSXL data component may expose the full flattened version of the DOM Level 2 interface to provide richer semantics accessing its data structures.
3. **UpdateControl**: (Optional) These operations are used to mark the start and end of group of updates. Use of this interface is optional, at the discretion of the container. Default behavior is to raise change notification events synchronously with each update.
4. **Validation**: (Optional) These operations are used to trigger evaluation of type constraints between the data instance and its model. They are a part of a data component supports the XForms model interface.
5. **Recompute**: (Optional) These operations are used to trigger evaluation of side effects from updates elsewhere in the data. Result is to call any handlers that are part of the internal implementation of the data component, i.e. those that will not be evaluated as a result of evaluating external control links (see control component below). These operations are also part of a data component supporting the XForms model interface.
6. **DataAvailability**: (Optional) These operations are used to signal the availability of the component's instance data. While the way in which the component connects to external data is not itself standardized, states associated with that data connection could be, such as unconnected, data requested, data partial, data available.

Notes

- WSXL defines a processing model, which may be implemented by a WSXL container, for applying updates to components it manages and for managing the steps required to flush updates to other components in the container bound with WSXL control components. The specific event handlers that implement updates among a container's components are defined using the control component.
- How might we implement states associated with data such as error, input, default, valid? How is this connected with the XML processing model, not just XFORMS processing model -- how does this connect with WSFL?
- It is an open issue whether XFORMS metadata, including relevance and required attributes, should be carried on a WSXL control component grouped together with the associated data component in a WSXL container, or on the control component that links data directly to presentation. We prefer a separate control component linked to data since if we put those model constraints on the arc linking presentation to the model, we will have to duplicate them for each presentation component we bind to the data.
- Starter set of components -- simple wrapper for web service, allows declarative creation of a WSXL data component; simple container for aggregating data components. Tooling could expose as drag/drop metaphor in visual editor.

2.2.3 Presentation Component

Synopsis

In addition to the WSXL base component interfaces described above, the WSXL presentation component implements interfaces used to describe and maintain DOM-accessible instances of presentation in WSXL applications. The WSXL presentation component is based on the UI specifications in XFORMS, and includes both XFORMS presentation instance and template functionality.

The WSXL presentation component does not specify any particular widget set. WSXL presentation components may consist of elements in arbitrary namespaces, and may generate output markup similarly in any target XML language. It is a goal for WSXL to be a foundational infrastructure for supporting particular XML widget sets for user interfaces as may emerge in the industry, including those in XFORMS, XUL, UIML, and so on.

Presentation Component Interfaces

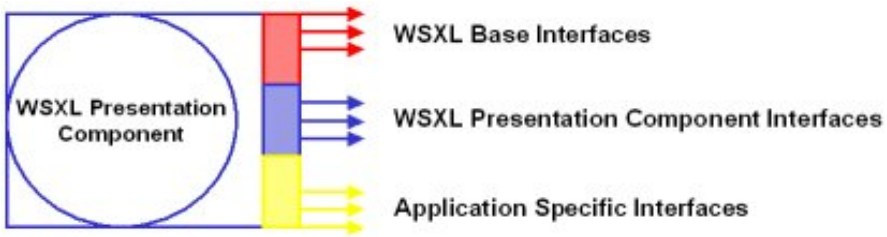

Figure 8: WSXL presentation component and interfaces

Beyond the base component interfaces, the additional categories of interfaces of a WSXL presentation component are shown in **Figure 8** and include:

1. **Access (Required)**: A WSXL presentation component describes the structure it wishes to expose using the WSXLDOMLite portType as defined above and in Appendix A.
2. **Interaction**: used to indicate state of end-user interaction with elements in the presentation component, such as focus, selection, activation, execution, hide/expose. Draws on and extends the abstract UI events in DOM Level 2 event model. Physical events such as mouse probably of less import to server-side components. Keyboard events required.

Notes

2.2.4 Control Component

Synopsis

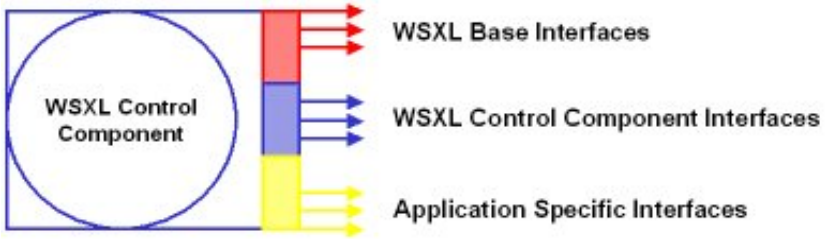

Figure 9: WSXL control component and interfaces

As shown in **Figure 9**, the WSXL control component is an extension of the WSXL component as are the data and presentation components. In addition to the base WSXL component interfaces, the WSXL control component implements interfaces used to manage the arcs binding data components to presentation components, to parse and interpret the XLINK-based control language specification described below, and to implement a processing model that controls the propagation of event notifications in both directions between data and presentation.

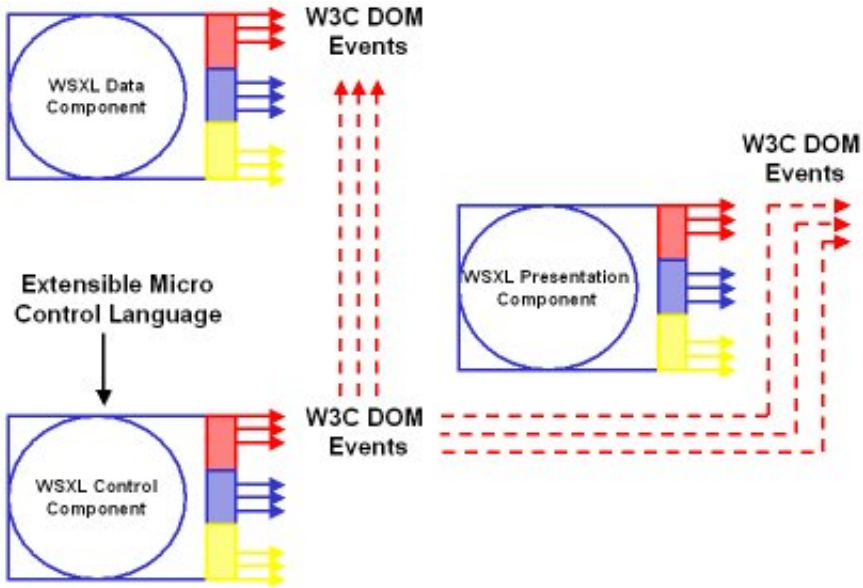

Figure 10: WSXL control component binding to WSXL data and presentation components

The basic control model is illustrated in **Figure 10**. The control component, either in response to the invocation of its arc management interface, or by parsing and interpreting a control language specification, establishes arcs between components, generally one being a data component and the other a presentation component. Each arc consists of an object which is a listener to a given element on the data component and to another element on the presentation component. Data and presentation components are not generally connected directly to each other. The control component establishes arcs and maintains itself at runtime to manage the order and timing of event notifications in each direction. Control component implementations may support directly connecting the endpoints of an arc, but will then need to deal with inserting themselves when operations to control the order and timing of event propagation are invoked.

Event handlers can be associated with arcs to provide control logic triggered in response to event notifications. This between-component, or "micro-level" control logic may (1) modify the values or structure of the data and/or presentation components, (2) make calls to external services, and (3) interact with across-component or "macro-level" control to trigger the possible instantiation of additional WSXL components for subsequent steps in the application.

It is important to note that micro and macro control logic are both triggered by events raised on arcs established and managed by a single type of WSXL control component. The distinction between them is simply in the nature of the changes effected by event handlers on those arcs. Event handlers that make changes within already instantiated and bound data and presentation components involve only micro-control logic. Event handlers that instantiate and bind new WSXL data and presentation components, or destroy existing ones and return to earlier steps in the application, involve macro-control logic. We expect that macro-logic may commonly be implemented through use of flow-language (e.g. WSFL) specifications invoked by event handlers on WSXL control components.

Control Component Interfaces

Beyond the base component interfaces, the additional categories of interface of a WSXL control component are shown in

Figure 9 and include:

1. Arc definition and management: (Required) These operations are used to create new arcs by providing source and destination component instances, XPATH locators within those instances, and event handlers to be associated with the arc. This interface supports operations to enumerate, create and destroy both locators and arcs. This interface can be called directly, or via the control specification language described below. The relevant standards for arcs include XFORMS, XLINK, and XML events.

Interface: *WSXLArcManagement*

getArcHandles(fromLocatorLabels, toLocatorLabels) - Returns an enumeration of handles to instantiated arcs.

getLocatorHandles(locatorLabels) - Returns an enumeration of handles to instantiated locators matching the supplied labels.

create / destroying locators - This functionality is provided through the createInstance / destroyInstance operations on the WSXLBase portType. The type to use for instantiating locators is “WSXLControl_Locator”. The name parameter can be used to specify the ‘label’ for the locator, but may be set as the ‘xlink:label’ property instead. The property ‘xlink:href’ is required as this specifies the URI for the locator.

create / destroy arcs - This functionality is provided through the createInstance / destroyInstance operations on the WSXLBase portType. The type to use for instantiating locators is “WSXLControl_Arc”. The properties ‘xlink:to’ and ‘xlink:from’ are required and must specify valid locators (either inline or by reference to defined locator labels). In addition, multiple properties named ‘event:listener’ of propertyType ‘Element’ may be supplied

batched creates - By setting a property named ‘Definitions’, a caller may supply a document similar to the example below and cause a set of locators and arcs to be instantiated.

Note: Will need to define an API for the arc and locator objects in order to make the handles returned useful. This API should include getting / setting the specification and destroy. An arc may also want to provide some statistical use information ….

2. Control specification parsing: (Required) The WSXLControl specification includes a specification language enabling compliant control components to parse an XLINK-based control specification file and create locators and arcs using the interface above. This interface supports a declarative way of defining the behavior of the control component without driving the underlying programmatic interface directly. A schema for this specification language in included in Appendix C.
3. Update processing model: (Optional) These operations are used to maintain list of “dirty” links, order list of updates and update groups, apply updates according to policies set by binding language (sync, async), fire event handlers on arcs. When this interface is not supported, events are propagated synchronously.

Interface: *WSXLArcControlProcessing*

queueEvents(sourceLocatorLabels) - This operation causes the control component to begin queueing events from the indicated locators. Passing null for the sourceLocatorLabels will cause all events to be queued for later processing.

releaseQueuedEvents(sourceLocatorLabels) - This operation will release any events queued for the indicated locators and cause events from those endpoints to no longer be queued. Passing null for the sourceLocatorLabels causes all events to be released and all event queueing stopped.

createLocatorGroup(groupName, locatorLabels) - This operation causes a group to be formed for the indicated locators. The groupName may then be used in subsequent calls to refer to all the locators within the group. Passing null for either the groupName or the locatorLabels list will cause a fault message to be returned.

Notes

- What is the need for multiple processing models? Portlets vs XFORMS vs batching, vs multi-modal? Will application authors in general define new processing models or should this be fixed? Perhaps in order to implement lazy vs greedy refresh?
- XFORMS does specify when sync should happen, and also defines the longest time when can be delayed. Checkpoints.
- Do the update groups require control component functionality, or are they just implemented by the data or presentation component? The update groups really correspond more to the UpdateControl concept above. This then requires an API within the Control component to support batches of updates.
- Control specification can bind to a number of different action languages: Java, Javascript, XML vocabulary like WSUI

2.3 Extensible Micro Control Language

The control component supports a declarative means of binding components of type data or presentation together. Already standardized syntaxes are extended as they already express most required concepts, namely; 1) XLink syntax provides a means for specifying 'locators' to name endpoints, which may exist in the same or different components, and 'arcs' which express a connection between a pair of locators and the semantics of traversing the arc (though the tranversal semantics need extension beyond that of navigation) and 2) XML Event syntax provides a means of specifying an action, its type, the event, event source and during what portion of event processing the action should be triggered. The following is a small example illustrating how these may be combined:

Presume the following is contained within www.examples.com/presentation.html:

```
<form id="Employee" . . .>
   <span> <div>First name</div>  <input name="firstname"></input>   </span>
   <span> <div>Last name</div>   <input name="lastname"></input>    </span>
   <span> <div>Employee #</div>  <input name="employeeNum"></input> </span>
</form>
```

Presume the following is a child of the root element in www.examples.com/data.xml:

```
<Employee>
   <Name>
      <First/>
      <Last/>
   </Name>
   <SerialNumber/>
</Employee>
```

Then an example Micro Control binding declaration could be:

```
<BinderLinks xmlns:ev="http://www.w3.org/2001/xml-events"
   xmlns:xlink="http://www.w3.org/1999/xlink"
   xlink:type="extended"
   xlink:title="Example binder link specification">

   <!-- First define the endpoints .... locators in XLink parlance -->
   <loc xlink:type="locator"
      xlink:href="http://www.examples.com/presentation.html"
      xlink:label="Presentation Root"/>

   <!-- 'parent' protocol indicates what follows is a locator's label -->
   <loc xlink:type="locator"
      xlink:href="parent://Presentation Root#//form[@id='Employee']"
      xlink:label="Employee Form"/>

   <loc xlink:type="locator"
```

```
         xlink:href="http://www.examples.com/data.xml"
         xlink:label="Data Root"/>

      <loc xlink:type="locator"
         xlink:href="parent://Data Root#/Employee"
         xlink:label="Employee Data"/>

      <!-- Now define the arcs between the endpoints -->
      <bind xlink:type="arc"
         xlink:from="Employee Data#Name/First"
         xlink:to="Employee Form#.//input[@id='firstname']/@value"/>

      <bind xlink:type="arc"
         xlink:from="Employee Data#Name/Last"
         xlink:to="Employee Form#.//input[@id='lastname']/@value"/>

      <bind xlink:type="arc"
         xlink:from="Employee Data#SerialNumber"
         xlink:to="Employee Form#.//input[@id='employeeNum']/@value">

         <!-- validate the entered serial number prior to writing it into the data
-->
         <ev:listener ev:event="DOMCharacterDataModified"
            ev:observer="xlink:to"
            ev:handler=http://testit#VerifySerialNumber
            handlertype="WebService"/>
      </bind>
   </BinderLinks>
```

These declarations start with two locators that each reference a URL through their xlink:href attributes. The Micro Control Component will ask the container that instantiated it to resolve these URLs. If the container has not already instantiated these, it will fetch the referenced component and locally instantiate it (or a proxy to it). The handle to local instance will then be returned to the Micro Control component.

A couple of additional locators are declared relative to these two root level locators. The Micro Control Component will query the components referenced by the parent locators to resolve the supplied query string (in this case an XPath).

The processing of the arc declarations will establish listener relationships between the referenced components and the controller using the DOM Level 2 Event API. The default event type is DOMCharacterDataModified with a default action making the value of the endpoints of the arc equal. As can be seen from the last arc definition, both the event and handler can be declared using XML Event syntax. An extension is required to indicate which endpoint is the source of the handled event (the example specifies the input field with id='employeeNum' in presentation.html). These handlers may format the data during the traversal of the arc by returning a new value, abort the traversal by returning null, as well as any desired side effects (e.g. trigger a transition in the Macro Flow between components).

2.4 WSXL Containers

Synopsis

A WSXL container provides an execution and management environment for WSXL components. It calls the lifecycle operations on WSXL components it instantiates, and implements a set of interfaces and a processing model for use by WSXL components and objects external to the container. An object implementing the WSXL container interface need not be a WSXL component.

Interfaces

A WSXL container defines two categories of interfaces: required and optional. The hasInterface operation from the WSXLBase interface may be used to query which optional interfaces the container implements. The container interfaces is divided into the following portTypes:

1. **Query:** (Required) A Container must provide standard means for querying the components and nested containers it is

managing. The simple query interface (modeled after DOM NamedNodeMap interface) required of all containers is shown below with the full normative WSDL form in Appendix A.

Interface: *WSXLQueryable*

getNameForHandle(handle) - Returns the name associated with the object instance when it was instantiated.

getNames() - Returns a document fragment enumerating all named handles.

getHandleCount() - Returns a count of the number of instantiated objects with handle references.

getHandles(selector, selectorType) - Returns a HandleList enumerating the handles matching the supplied selector. Examples of selectorTypes include ‘name’ (the selector is the name associated with the returned handle) and ‘xpath’ (the selector is an xpath statement used to select the desired list of handles).

2. **Hosting:** (Required) Container must provide a means to specify the components that are to be instantiated. The container instantiates the component and invokes operations on the component that are part of the initialization/lifecycle contract. Rather than defining an additional set of operations, setting a property named “DefinitionsDocument” will result in that document being parsed and the components / nested containers specified by it being instantiated. An example of a DefinitionsDocument and the schema for them can be found in Appendix B.
3. **Management:**(Optional) Once a component is instantiated, a container may provide an interface that allows the component to request resources. An example of such a resource is display "real estate". A container in turn decides when to remove a component and deallocate its resources.
4. **Environment and Context service:** (Optional) This interface deals with how a component can find out details about the environment. Here the environment encompasses a) the details of specific configuration and the deployment platform in which the component has been instantiated, b) the details and objects of the specific session/context in which the component is being executed. The latter item includes APIs that deal with personalization (device information, user profile) etc.
5. **Event service:** (Optional) A container may support the WSXLEventSource and WSXLEventSink interfaces. Supporting these interfaces allows a component to register interest and handlers for events without having to know the existence of all components that may generate that event. This also provides decoupling between an event generator and event listener with the container acting as a broker that controls the dispatch and propagation of an event.
6. **Component discovery and invocation:** (Optional) A container may offer an API where a component implementation or instance (local or remote) can be found at runtime based on some attributes of the component interface. Then the hosting API can be used either to instantiate the discovered component locally, or the component may establish a separate direct binding to the discovered component.

Notes

- Making the requirements on a container interface minimal results in making the container writer's job easy. However, does it make the WSXL component writer's job hard? What tradeoff should we make?
- On a related note, should we subdivide the categories more to have more required and optional categories?
- Given the definition of a container, an application is a container. Wrt Roland's comment, we need to decide where at what level of discussion we want to give. From the flow perspective, Rich and I think it is just another subsection under the description of the containers.

Relevant Standards

The newly started working group on XML plugins.

Other standards/practices for conceptual input, some of them not in the markup domain, are HTML behaviours, netscape plug-in API, EJB Containers. Java AWT.

2.5 Creating Applications from WSXL Components

Given the definitions of WSXL components and containers, how does one write a web application in terms of these objects? The simplest web application built out of WSXL components is a single base component, which may in turn be implemented

in any way desired, including through the use of other WSXL components. Beyond this, an application may be a WSXL container that can have one or more WSXL components, e.g., one or more of presentation data, and control components with bindings and event handlers. Applications can be aggregated inside WSXL containers. An application that further desires to be a WSXL component itself, i.e, a reusable type, needs to implement the WSXL Component interfaces as well, thus becoming a valid WSXL Component amenable for instantiation and extension.

2.5.1 Reusable Groupings of Components

By allowing groupings of data, presentation, and control instances in a WSXL Container we hope to simplify the creation of application *instances*, i.e. code that by not being required to support reuse is simpler to implement by non-programmers. By further allowing groupings to implement the WSXL Component interface, we provide a simple composition framework for creating larger units of an application.

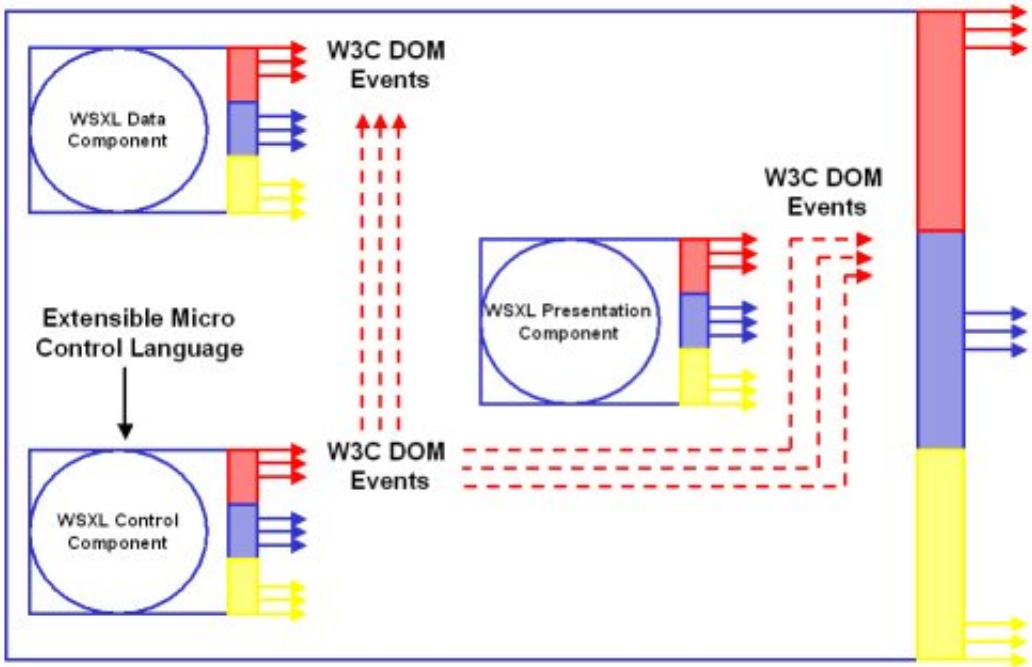

Figure 11: WSXL aggregate component built from a combination of WSXL data, presentation, and control components

The object implementing the WSXL Container describe in the section above may further implement the WSXL Component interfaces to allow the grouping to be reused together with other WSXL components. **Figure 11** shows the interfaces that are exposed by such aggregate WSXL Components. As is true of all WSXL Components, this interface includes the WSXL Base interfaces (Lifecycle, Access, and Events), along with any nested data, presentation, control, and application specific interfaces that are re-exposed by the grouping.

Note that in this discussion even though we use the terms data and presentation in the singular, and show them as single icons in the figure, in general either data or presentation may consist of multiple instance fragments -- the controller supports reference to any number of fragments on either the data or presentation side.

Components may be instantiated, and these instantiations assembled to create new aggregations. Such aggregations can be created from instances of different types, e.g., as in the earlier data/presentation example, or from instances of the same type, e.g. aggregating multiple data instances. For example, two data components being aggregated using a WSXL control component to produce an aggregated data component. The creator of such an aggregation can in turn implement the Component interface to enable this new aggregation to be used as an atomic data component in another composition such as the one above in **Figure 11**.

2.5.2 Multi-modal Coordination of Components

Multiple presentation components might be grouped with a control component in order to support synchronization across multiple active presentations -- whether on the same or different devices. Presentation components might also be aggregated to select from among them rather than to synchronize their function. In this case, the control component can act as a selector in addition to being a combiner. For example, a set of presentation components might be grouped together along with a control component to determine which among them is the active frame. Seen from the outside, the group functions simply as a presentation component which can bind to data and control components just as shown above in **Figure 11**.

2.5.3 Flow of Control Across Components

Previous sections of this document have focused on describing the interfaces of various types of WSXL components, such as Data, Presentation, and Control. Containers have also been discussed in terms of hosting these components and providing services to them. A Container that instantiates a particular set of data, presentation, and control in order to implement a single function is loosely analogous to a "page". One of these page-level Containers could execute as a stand-alone application, even though a single page application may not be particularly useful.

However, if a page-level Container also implements the Component interface, it can be composed into a Container. In fact, several of these page-level Container/Components can be grouped together into a Container. This Container is much more interesting as an application since it has more than one "page".

The interesting question to answer at this point is how does a Container define and execute its behavior? How is the navigation between and among Components defined? The WSXL architecture defines the Control Component as having this responsibility. WSXL does not specifically provide a language for describing macro control, but rather relies on a pluggable architecture that allows a variety of implementations. Some examples of macro control implementation in a Container are:

1. Procedural code such as Java that initializes and executes components in the desired order.
2. A rules engine.
3. A WSFL flow engine.
4. A state machine.

Depending on the strategy chosen, a Container could externally publish its behavior. For example, a Container could publish its behavior in terms of a WSFL flow model. This would allow queries both during development and at run-time. Development-time queries on a macro flow may be useful as a form of documentation for a programmer developing a cooperating application. Run-time queries on a macro flow may be useful for determining allowable next steps. Furthermore, an architecture that supports externally defined control flow allows applications to be very agile with respect to changing requirements.

While WSFL is intended to publish and orchestrate the interactions of web services with each other in order to accomplish a particular task, we are particularly interested in the definition of flow models. A flow model is a directed, a-cyclic graph, which describes a business process. The business process is described in terms of activities that are connected by data links and control links. These links connect activities together and thereby define their order of execution. Conditional expressions on control links are used to implement conditional branching among the activities in the graph. Another interesting point about flow models is that they can be recursively composed. Therefore, a flow model can invoke another flow model as a child activity. This enables a functional decomposition of an application that promotes re-use.

In a WSFL flow model, an activity represents an operation on a web service. WSFL flow models can be applied to the Macro Control Component if we establish that an activity maps to an operation that starts and executes a Component. The flow model can therefore be used to describe the order of Components within the Container as well as conditional branching among them. Since WSFL is meant to orchestrate web services, this applies nicely to WSXL because each Component has a well-defined WSDL lifecycle interface. Thus, an activity defined on a flow model can invoke the operation that kicks off the Component. During the Component's execution, the Control Component determines the completion status of the Component via its event handlers. The flow model evaluates completion status for the Component (activity) to determine whether to repeat the activity or proceed to the next activity in the flow.

2.6 Adaptation Description Language

One of the issues in application syndication is that a component provider may want to have some say in how the markup and operations provided by the component may be accessed, modified, or overridden. This need may be for brand preservation, to preserve a certain aesthetic design constraints, or for correctness reasons. Thus a "conforming" client/aggregator can use the information to reliably adapt the component output and to intercept user-input to the component within the constraints laid down by the provider. To provide infrastructure support for such adaptation, we propose a machine oriented means of specifying this adaptation information i.e., not plain english or pick-your-spoken-language.

To this end, WSXL includes an extensible Adaptation Description Language where explicit locations of adaptation points, the permissible operations on adaptation points (e.g. insert, delete, modify, ...), and the constraints on the contents of adaptation (e.g. via an XML Schema) can be specified. The Adaptation Description Language can be used during a

post-processing step where the output of a WSXL component can be adapted independently without invoking the component.

The core concept of this description language is the notion of an Adaptation point. Adaptation points are a means of specifying how a particular user experience can be adapted to the channel through which it is delivered to the end user. Adaptation points define "observable characteristics" of a presentation stream, and as such, they are analogous to and complementary to message types. Adaptation points are used at design time, by both the application provider and the intermediary, to speed development and help ensure correctness. Further, they are used at runtime, by supporting middleware, to safely perform the adaptations requested by an intermediary. For example, a provider can specify Adaptation points that allow a client use the information to reliably:

- Change background color or font variable used by the application [Presentation adaptation]
- Lookup a value in the output markup (e.g. an embedded product ID) [Presentation adaptation]
- Overwrite/Add to a value in the output markup (e.g. a price) [Data adaptation]
- Insert a value in a particular page (e.g. an extra text field, a new column in a table) [Presentation and possibly Control adaptation]
- Override/extend the action associated with an event (e.g. button press) [Control adaptation]

Adaptation Points are part of a WSXL component definition. Each Adaptation Point specifies:

- An application-specific Adaptation Point name
- The kind of operation (insert, replace, lookup). Specifies the adaptation's intended use.
- The names of pages, i.e., to the different component's outputs, that it applies to. Optional. This information is used at design time and in low cost runtime implementations.
- A locator to the item[s] of interest. For instance, xpath, xquery are used for XML documents. Note that a locator can point to multiple locations in document.
- An Adaptation Point category (XML, CSS, etc). Specifies the category of the item[s] of interest.
- The information specific to the application of an adaptation of that category.
- Tooling Hints

As part of defining ports in the context of a service definition, a WSXL port can optionally specifiy an attribute named adaptationURI. An example of specifying the adaptationURI is as follows:

```
    <service name="LeatherShopService">
        <documentation>My first service</documentation>
        <port name="LeatherShopPort" binding="LeatherShopBinding"
adaptationURI="http://TLS.com/LSAdaptation.xml" />
           <soap:address location="http://TLS.com/shop"/>
        </port>
    </service>
```

The adaptationURI refers to the adaptation description file for the port. The root element of the adaptation description language is an Adaptation group which can consist of a combination of Adaptation Groups and Adaptation Points. AdaptationGroups are elements solely for application-specific organization of exposing adaptation points. The following categories of adaptations are initially introduced:

- CSSAdaptation,
- XMLAdaptation,

These two have been introduced to handle the manipulation of two typical contents of messages i.e., CSS and XML. Other categories can be introduced into the language as necessary.

To allow a provider to assist or control the markup inserted by an aggregator, a concept of adaptation generator is introduced. An AdaptationGenerator is an operation on the provider service that can be invoked by clients to request customized markup for inclusion in the provider service's output message stream.

2.6.1 XMLAdaptation

XMLAdaptation elements have been introduced to accommodate the specification and manipulation of XML. This element can appear as part of defininig an adaptation and used to provide information related to manipulating XML fragments.

Several examples are shown below:

```
<adaptation name="CustomerNumber" use="lookup">
   <locator xpath="/xhtml/body/table/row/customer">
   <XMLAdaptation>
      <validTypes>integer</validTypes>
   </XMLAdaptation>
</adaptation>
```

In the above example, the provider is specifying an adaptation for lookup for data items present in the output stream of a presentation component and indicating that the type of the value would be an integer.

```
<adaptation name="ContactInfo" use="replace">
   <locator name="ContactLocator" type="xpath" xpath="//cntctInfo"/>
   <XMLAdaptation>
      <validType> ShortContactInfo LongContactInfo </validType>
      <schemaLocation>http://www.SampleCompany.com/TypeInfo.xml </schemaLocation>
   </XMLAdaptation>
</adaptation>
```

In the above example, the provider is specifying an Adaptation Point to allow the replacement of the contact information in its output stream with the constraint that the replacing XML fragment should match one of the two specified types. The type description is available in the file at the URL specified in the schemaLocation element.

As the last example, a data component provider may wish to allow a client to replace an item's price in dollars by an equivalent price in different currencies. However, the provider wishes to express a constraint that the price inserted should obey certain constraints.

```
<adaptation name="PriceColValues" use="replace">
   <locator xpath="//PriceOfTheItem" type="xpath" >
   <XMLAdaptation>
      <validTypes> PartPriceType</validTypes>
      <schemaLocation> http://www.SampleCompany.com </schemaLocation>
   </XMLAdaptation>
</adaptation>
```

2.6.2 CSSAdaptation

CSSAdaptation elements have been introduced to accommodate the specification and manipulation of CSS attributes that do not conform to XML. An example instance of CSSAdaptation along with its enclosing adaptation and adaptationGroup is shown below:

```
<adaptationGroup name="changeLookAndFeel">
   <adaptation name="backgroundColor" use="replace">
      <outputList> output1 output3 output10 </outputList>
      <locator name="backgroundColorLocator" type="xpath"
xpath="xhtml/body//table"/>
      <CSSAdaptation>
         <property> background-color</property>
      </CSSAdaptation>
      <comment>If you have problems, contact Popeye. </comment>
      <toolingHint>Specify override for background color. Default is green
</toolingHint>
   </adaptation>
</adaptationGroup>
```

In this description fragment, the adaptationGroup has only one Adaptation Point, which provides information to aggregator about how the background color of a XHTML table can be adapted. The name attributes have only human significance. The

"use" attribute on the adaptation specifies that a client can perform a replace operation on the item specified within the adaptation element. The outputList element consists of names that identify the subset of the different outputs to which the Adaptation Point applies. The locator element specifies where in the component's XML portion of the output the item of interest can be found while the CSSAdaptation specifies the CSS qualifying locator that identifies the specific attribute of interest. In this case the values in the property element must match one of the properties in the CSS specification.

Note that CSSAdaptation are for dealing with inline CSS attributes only. All other "out-of-band" CSS specifications can be adapted by taking advantage of CSS specification by inserting a pointer to the CSS style sheet as "close" to item of interest as possible and leaving it to mechanics of the CSS applicator to apply them correctly.

2.6.3 AdaptationGenerator

AdaptationGenerators have been introduced to handle three kinds of situations:

1. For providers to provide assistance in creating markup for the aggregator for use in adaptation.
2. To retain control over markup fragements that are inserted
3. To sidestep the limitations of the constraint language.

A provider service implements these AdaptationGenerator operations and includes their names as part of the AdaptationGenerator specifications. These operations are invoked by a client to perform presentation related transforms and use the resulting message in adaptation. Here is an example of a provider implementing an operation named "addColumnToXHTMLTable" and includes it within an AdaptationGenerator construct.

```
<adaptation name="addColumnAtEndOfTableA" use="replace">
   <locator name="TableALocator" type="xpath" xpath="//..."/>
   <AdaptationGenerator>
      addColumntoXHTMLTable
   </AdaptationGenerator>
</adaptation>
```

The exact signature of the name specified in an AdaptationGenerator is obtained by looking at the WSXL specification of the service. It is expected to have an input message that consists of two parts: a) a portion of the message originally sent down by the provider or locators that point into the message, b) the presentation and/or content that the client wishes to be added to the provider's message portion. The output signature would be a that of a message that can contain any instance that can accomodate the client's content.

3 Web Services For Remote Portals (WSRP)

WSRP's Remote Portlet Web services are visual, user-facing web services centric components that plug-n-play with portals or other intermediary web applications that aggregate content or applications from different sources. They are designed to enable businesses to provide content or applications in a form that does not require manual content or application-specific adaptation by consuming intermediary applications. As Remote Portlet Web wervices include presentation, service providers determine how their content and applications are visualized for end-users and to which degree adaptation, transcoding, translation etc. may be allowed.

Remote Portlet Web services can be published into public or corporate service directories (UDDI) where they can easily be found by intermediary applications that want to display their content. Web application deployment vendors can wrap and adapt their middleware for use in WSRP-compliant services. Vendors of intermediary applicatios can enable their products for consuming such services.

Using WSRP, portals can easily integrate content and applications from many internal and external content providers. The portal administrator simply picks the desired services from a list and integrates them; no programmers are required to tie new content and applications into the portal.

To accomplish these goals, the Remote Portlet Web services standard defines a web services interface description using WSDL and all the semantics and behavior that web services and consuming applications must comply with in order to be pluggable as well as the meta-information that has to be provided when publishing Remote Portlet Web services into UDDI directories. The standard allows Remote Portlet Web services to be implemented in very different ways, be it as a Java/J2EE based web service, a web service implemented on Microsoft's .NET platform or a portlet published as a Remote Portlet Web service by a portal. The standard enables use of generic adapter code to plug in any Remote Portlet Web Service into

intermediary applications rather than requiring specific proxy code.

Remote Portlet Web services are WSXL component services built on standard technologies including SOAP, UDDI, and WSDL. WSRP adds several context elements including user profile, information about the client device, locale and desired markup language passed to them in SOAP requests. A set of operations and contracts are defined that enable WSRP plug-n-play.

3.1 Remote Portlet Web Services and Portals

While Remote Portlet Web Services can be consumed by different kinds of intermediary applications, they are of particular importance for portals. Portals are focal points for users to access information and applications from many different sources. Typically, portals get information from local or remote data sources, render and aggregate this information into composite pages to provide information to users in a personalized and easily consumable form. In addition to pure information, many portals also include applications like e-mail, calendar, organizers, banking, bill presentment, etc. All of these application components rely on the portal's infrastructure and operate on data or resources owned by the portal, like user profile information, persistent storage or access to managed content. Consequently, most of today's portal implementations provide a component model that allows plugging components referred to as Portlets into the portal infrastructure.

3.2 Publishing, Finding and Binding Remote Portlet Web Services

In order to make remote portlet web services discoverable for clients, Remote Portlet Web Services can be published to a UDDI directory. In order to make remote portlet web services known, the provider must publish them to a UDDI directory. To allow for any remote portlet web service to plug into any compliant client application, all remote portlet web services implement the same WSDL interface and behavioral contracts. Furthermore, they share a common set of attributes that can be analyzed by portals to find out what capabilities the service provides and how it has to be used by client applications. Remote Portlet Web services can be found in UDDI directories by querying the directory for businesses providing services implementing the Remote Portlet Web Services tModel and listing the Remote Portlet Web services provided by particular businesses.

3.3 WSRP and WSXL

Remote Portlet Web Services defined in WSRP are special WSXL components, sharing the basic interfaces and contracts defined in WSXL. This means that Remote Portlet Web Services behave like other WSXL components in terms of life cycle and provide operations for providing markup and for processing user actions. While WSXL defines a general programming model for web service-based application components and base interfaces, Remote Portlet Web services can be seen as an application of WSXL, defining a concrete interface and contract for components intended to be aggregated in and driven by client applications like for example portals.

By standardizing the complete interface and behavior for compliant services and client applications in addition to what is standardized in WSXL, the Remote Portlet Web Services specification enables plug-n-play interoperability between any compliant client application and Remote Portlet Web services, so that client applications can invoke Remote Portlet Web Services always using generic invocation code.

WSXL provides a generic set of basic interfaces for life cycle, presentation and event handling. WSRP implements and extends selected WSXL interfaces and provides a specialized interface to its services. WSRP is designed so that implementing a compliant service is as easy as possible, services are extensible and interfaces can be efficiently accessed. For interoperability, WSRP services not only provide specialized and efficient access through the WSRP interface for WSRP clients like portals but also provide generic access through the WSXL interfaces for WSXL clients. We envision the following relation of WSRP and WSXL:

1. WSRP services must implement the basic WSXL interfaces
2. WSRP services must implement the WSXL markup interfaces
3. WSRP services should implement the WSXL property interfaces
4. WSRP services may implement the WSXL event interfaces

4.0 Web Services Experience Language By Example: Implementation Details

In this section we revisit the examples from the start of this paper, and demonstrate how WSXL can be used to implement them.

4.1 Syndicated Purchasing of Traveler Checks

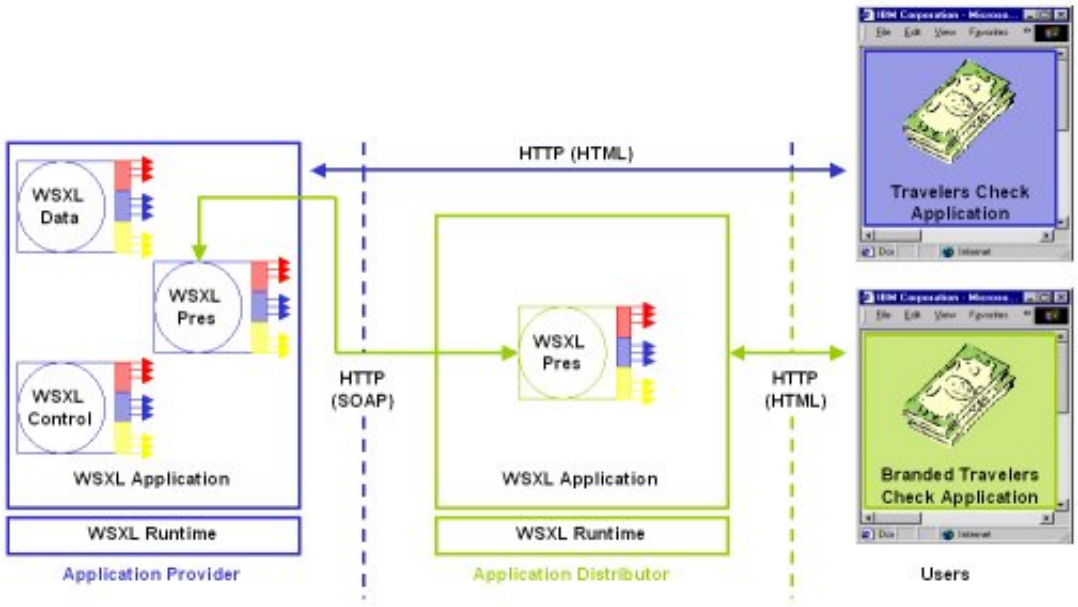

Figure 13: WSXL Implementation of Syndicated Purchasing of Traveler Checks

As depicted in **Figure 13**, the provider can describe a WSXL application that can then be used to serve direct client access or as the business logic within a remote application re-distributed by a business partner. In the case of the later, the distributor adds an additional WSXL Presentation Component with two functions: (1) to adapt the presentation of the provider's application to the requirements of the distributor, and (2) to add any additional presentation elements required for new functionality introduced by the distributor.

In this scenario, the distributor simply establishes an alternative brand view of the application for its clientele by varying the skin of the provider's presentation. The presentation component at the distributor level, therefore, generates its own output markup perhaps by applying different XSLT or CSS style sheets.

Besides achieving application syndication with WSXL for presentation purposes, the provider's application can also have direct control over events raised by interactions at the browser. In implementation terms, the distributor's presentation component delegates all events received from the client to the provider for interpretation within the provider's control and data context.

4.2 Distributed Visualization Services (Local Portal)

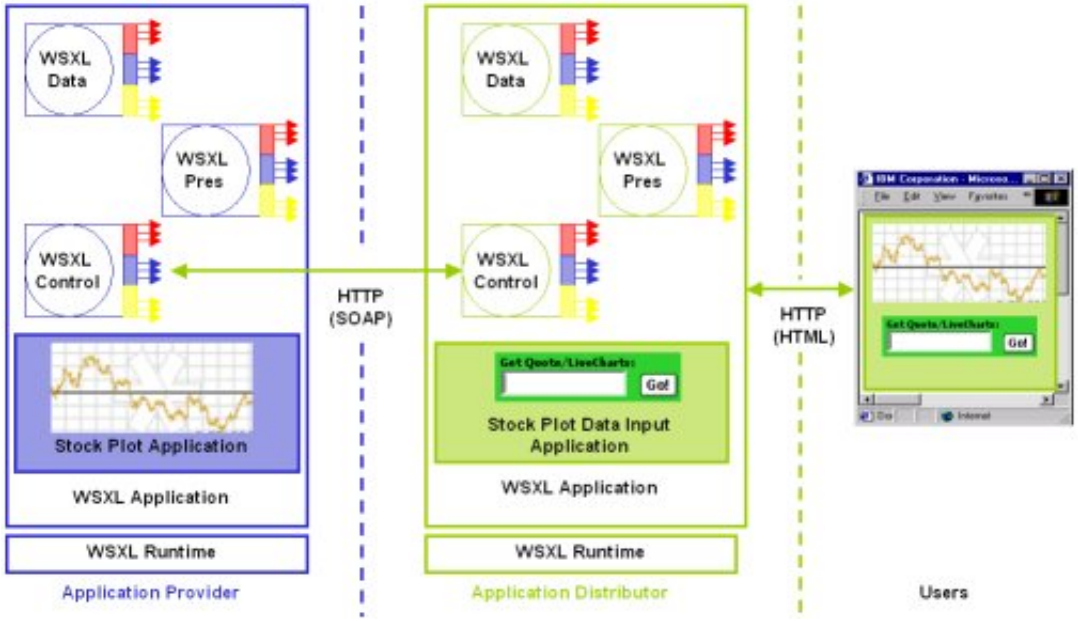

Figure 14: WSXL Implementation of Distributed Visualization Services (Local Portal)

Building on the prior scenario, **Figure 14** demonstrates how a portal provider could also OEM their portal product that contains access to the financial graphics utilities. In this case the distributor in the application supply chain is adding functionality beyond simply altering the look and feel of the provider's application. Additional Presentation and Data components are introduced at the distributor to create the data entry form required for the stock symbol to be charted. A Control component at the distributor links the data and presentation elements for the stock symbol, and carries an event handler on that arc for passing the entered data value to the provider's application control component to trigger the regeneration of a chart. From the standpoint of the end user there is a single application frame whose events are coordinated and delegated to the appropriate providers by the distributor.

If implemented as a local portal, as described in this section, the presentation, data, and control components introduced by the distributor can be managed by a simple WSXL Container. In order to support remote access through a second tier distributor -- i.e. as a remote portal -- then this grouping would in turn implement the WSXL Component interface to allow for remote binding and interaction.

4.3 Value Added Reselling of Traveler Checks

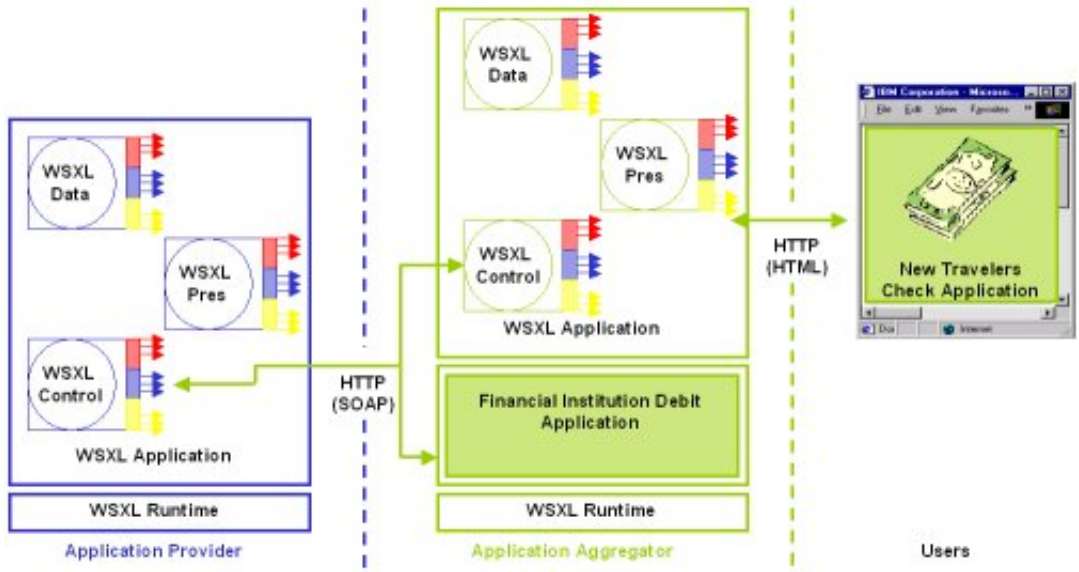

Figure 15:WSXL Implementation of Value Added Reselling of Traveler Checks

Our last scenario, depicted by **Figure 15**, suggests how WSXL could be used to create new revenue building applications. In this case the application domain represents the selling of Traveler Checks. However, the bank in this case is a value added

reseller since it seeks to enhance existing applications assets of their business partner (the Traveler Cheque Supplier) with a new payment instrument, specifically debit.

In order to achieve this goal the bank creates customized presentation layer by integrating the remote application of its business partner with a locally available application that addresses the handling of debit transactions from bank member accounts. The local application consists of a WSXL Presentation component to capture the specific user information required for the debit transaction. The local WSXL Control component then contains an event handler to pass this information to a bank-hosted debit application implemented in whatever technology is appropriate for the bank. Since the debit transaction does not involve further end-user interaction it is itself not a WSXL application. Note, however, that the WSXL Control component that initiates this debit is required also to notify the provider's Control component that a transaction has been initiated so that the provider's application can proceed to the appropriate next step of acknowledgement or termination.

4.3 WSXL and Macromedia Flash

WSXL components can live on any tier of the network and are independent of output markup.

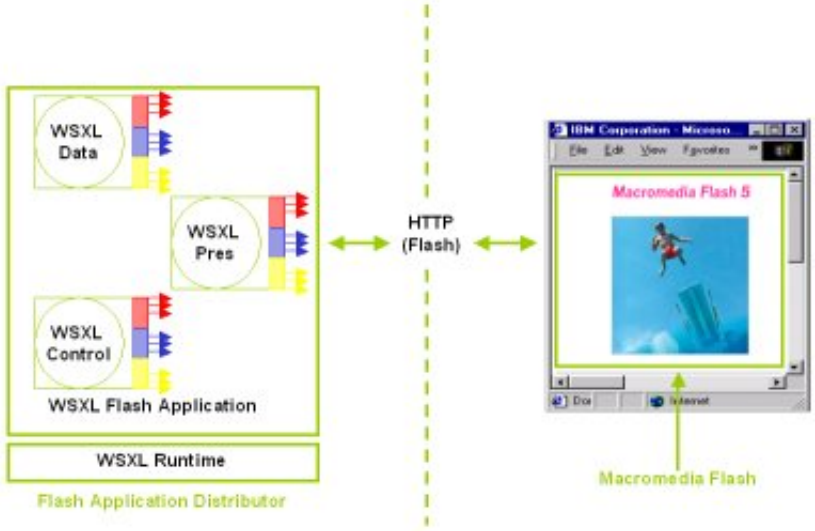

Figure 16:WSXL and Flash markup

Figure 16 demonstrates a WSXL application producing Macromedia Flash content.

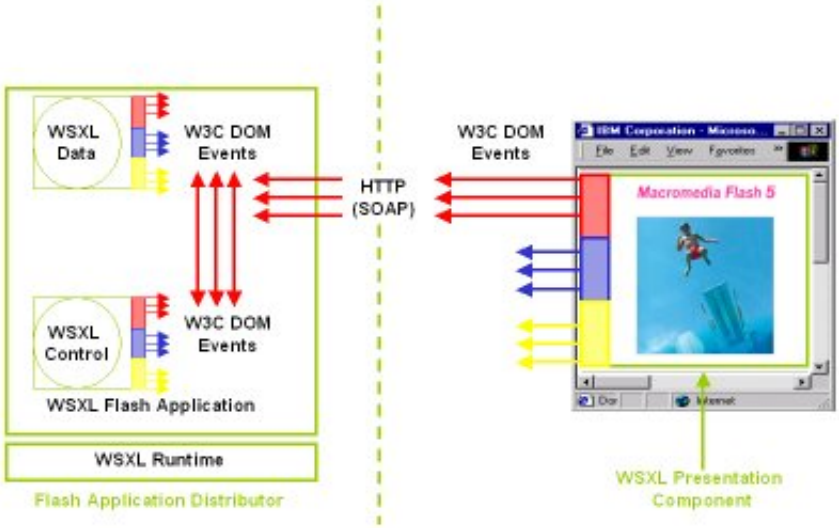

Figure 16:WSXL and Flash Presentation Component

In the example shown in **Figure 16**, the WSXL Presentation component is moved to the client in order to support direct generation of presentation using the Flash runtime engine. The Flash runtime is implemented inside a WSXL Flash Presentation Component as an extension of the base WSXL Presentation Component. As such, the WSXL Flash Presentation interfaces include the base WSXL Component interfaces for DOM access and events (shown in red in the Figure), and the standard WSXL Presentation interfaces (shown in blue). The WSXL Flash Presentation component defines additional

interfaces (shown in yellow) to support Flash specific functions such as display and physical interaction with the user.

WSXL components may execute on any tier of the network, and use W3C DOM Events to signal changes between them through arcs established by WSXL Control Components. In the figure, the WSXL Flash Presentation Component interacts with the remainder of a WSXL application through control links established by a WSXL Control Component. Data can be provided to the Flash presentation, and results returned from it, by receiving and raising events on these links. Changes in the interaction state of the flash presentation can also be signaled using events in order to synchronize the behavior of the flash presentation with non-flash parts of the user interface that may also be present.

5 Summary

WSXL is the next piece of the web services stack. WSXL provides a web services standards based approach for web application development, deployment and maintenance. WSXL enables *Dynamic e-Business*, by moving from transactions driven by a single business entity to a world of electronic transactions involving multiple business entities who compose and aggregate re-usable web applications. As a result, these applications can leverage new and innovative revenue models.

6 Appendix

6.1 Appendix A: WSDL Descriptions

6.1.1 WSXLBindable

WSXLBindable interface describes objects that support a separate WSDL binding [WSXLBindable-20020101.xml].

6.1.2 WSXLBase

WSXLBase interface includes WSXLBindable and general lifecycle operations [WSXLBase-20020101.xml].

6.1.3 WSXLProperties

WSXLProperties interface includes property management operations [WSXLProperties-20020101.xml].

6.1.4 WSXLMarkup

WSXLMarkup interface provides a means for a client to request an object's markup [WSXLMarkup-20020101.xml].

6.1.5 WSXLEventSource

WSXLEventSource interface includes event source operations [WSXLEventSource-20020101.xml].

6.1.6 WSXLEventSink

WSXLEventSink interface includes event sink operations [WSXLEventSink-20020101.xml].

6.1.7 WSXLComponent

WSXLComponent interface includes WSXLBase, WSXLEventSource, WSXLEventSink and the getMarkup operation [WSXLComponent-20020101.xml].

6.1.8 WSXLQueryable

WSXLQueryable interface includes simple query operations [WSXLQueryable-20020101.xml].

6.1.9 WSXLContainer

WSXLQueryable interface includes WSXLQueryable and WSXLBase [WSXLContainer-20020101.xml].

6.1.10 WSXLArcManagement

WSXLArcManagement interface includes interfaces for managing Control Language Arcs [WSXLArcManagement-20020101.xml].

6.2 Appendix B: Container Definitions Document

6.2.1 Example of a Container Definitions Document

```
<WSXLContainer name="StockChartContainer"
xmlns:ev="http://www.w3.org/2001/xml-events">

  <WSXLComponent name="ChartUI" role="Presentation"
href="http://www.StockRUs.com/StockChart.wsxl"/>

  <WSXLComponent name="ChartData" role="Data"
href="http://www.StocksRUs.com/StockFeed.wsxl" >
    <PropertyList>            <!-- default propertyType="String" -->
      <Property propertyName="DataService" propertyValue="Tipco"/>
      <Property propertyName="StockSymbol" propertyValue="IBM"/>
      <Property propertyName="TimeFrame"   propertyValue="1 day"/>
    </PropertyList>
  </WSXLComponent>

  <WSXLComponent name="ChartController" role="Control"
href="http://www.StocksRUs.com/ChartController.wsxl">
    <PropertyList>
      <Property propertyName="ContainerContext"
propertyValue="name://StockChartContainer"/>
      <Property propertyName="ControlFile" propertyType="file"
propertyValue="http://www.StocksRUs.com/ChartController.xlink" />

      <!-- if more than one control file is needed, just repeat the ControlFile
property like so:
      <Property propertyName="ControlFile" propertyType="file"
propertyValue="http://localhost/StockChartContainer/controlSettings2.xlink" />
      -->
    </PropertyList>
  </WSXLComponent>

<ev:listener event="ContainerInited" handler="ChartData-InitHandler"
type="javascript"/>

</WSXLContainer>
```

6.3 Appendix C: Control Specification Language

6.3.1 Control Specification Language Schema

[WSXLControlLanguage-20020101.xsd].

6.4 Appendix D: WSXL and WSRP Dependency Diagram

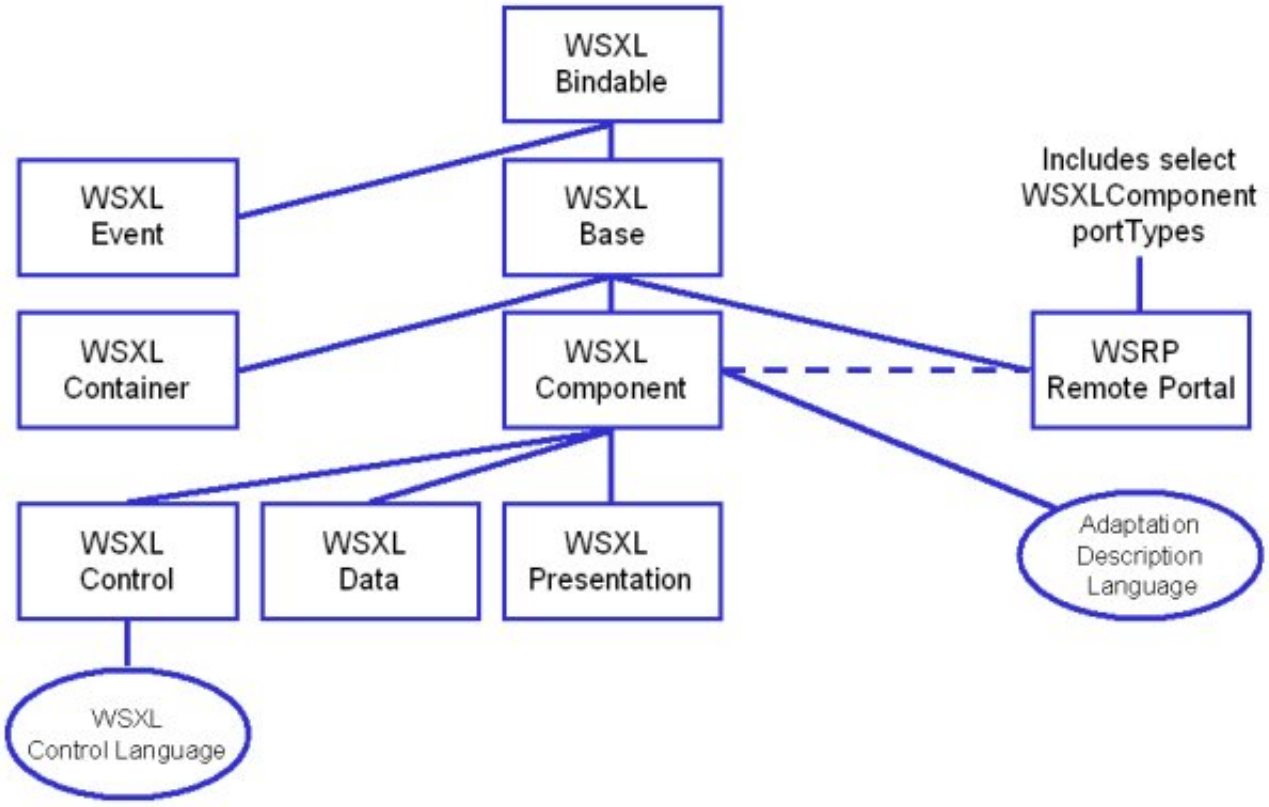

Figure 16:WSXL and WSRP Dependency Diagram

7 References

DOM Level 1

W3C (World Wide Web Consortium) DOM Level 1 Specification, October 1998. Available at http://www.w3.org/TR/REC-DOM-Level-1

DOM Level 2 Core

W3C (World Wide Web Consortium) Document Object Model Level 2 Core Specification, November 2000. Available at http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113

DOM Level 3 Events

W3C (World Wide Web Consortium) Document Object Model Level 3 Events Specification, August 2001. Available at http://www.w3.org/TR/DOM-Level-3-Events

DOM Level 2 HTML

W3C (World Wide Web Consortium) Document Object Model Level 2 HTML Specification, November 2000. Available at http://www.w3.org/TR/2000/WD-DOM-Level-2-HTML-20001113

HTML 4.0

W3C (World Wide Web Consortium) HTML 4.0 Specification, April 1998. Available at http://www.w3.org/TR/1998/REC-html40-19980424

RFC2396

IETF (Internet Engineering Task Force) RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax, eds. T. Berners-Lee, R. Fielding, L. Masinter. August 1998. Available at http://www.ietf.org/rfc/rfc2396.txt

XML Events

W3C (World Wide Web Consortium) Title, availability date, and URL not available at press time.

XForms

W3C (World Wide Web Consortium) XForms, August 2001. Available at http://www.w3.org/TR/xforms

XLink

W3C (World Wide Web Consortium) XLink, June 2001. Available at http://www.w3.org/TR/xlink/

XPointer

W3C (World Wide Web Consortium) XML Pointer Language (XPointer), January 2001. Available at http://www.w3.org/TR/xptr

XSLT

W3C (World Wide Web Consortium) XSLT, January 2001. Available at http://www.w3.org/TR/xslt

XML

W3C (World Wide Web Consortium) Extensible Markup Language (XML) 1.0, October 2000. Available at http://www.w3.org/TR/2000/REC-xml-20001006

XML Namespaces

W3C (World Wide Web Consortium) Namespaces in XML, January 1999. Available at http://www.w3.org/TR/1999/REC-xml-names-19990114