UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

A FRAMEWORK FOR RAPID INTEGRATION OF PRESENTATION COMPONENTS

Jin Yu, Boualem Benatallah, Regis Saint-Paul, Fabio Casati, Florian Daniel and Maristella Matera

December 2006

Technical Report # DIT-06-100

A Framework for Rapid Integration of Presentation Components

Jin Yu, Boualem Benatallah,
Regis Saint-Paul
University of New South Wales
Sydney, Australia

{jyu,boualem,regiss}
@cse.unsw.edu.au

Fabio Casati
University of Trento
Trento, Italy

casati@dit.unitn.it

Florian Daniel,
Maristella Matera
Politecnico di Milano
Milano, Italy

{daniel,matera}@elet.polimi.it

ABSTRACT

The development of user interfaces (UIs) is one of the most time-consuming aspects in software development. In this context, the lack of proper reuse mechanisms for UIs is increasingly becoming manifest, especially as software development is more and more moving toward composite applications. In this paper we propose a framework for the integration of stand-alone modules or applications, where integration occurs at the *presentation* layer. Hence, the final goal is to reduce the effort required for UI development by maximizing reuse.

The design of the framework is inspired by lessons learned from application integration, appropriately modified to account for the specificity of the UI integration problem. We provide an abstract component model to specify characteristics and behaviors of presentation components and propose an event-based composition model to specify the composition logic. Components and composition are described by means of a simple XML-based language, which is interpreted by a runtime middleware for the execution of the resulting composite application. A proof-of-concept prototype allows us to show that the proposed component model can also easily be applied to existing presentation components, built with different languages and/or component technologies.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces, Software libraries.* H.5.2 [**Information Interfaces and Presentation**]: User Interfaces – *Graphical user interfaces, Interaction styles, Prototyping, Standardization.* H.5.4 [**Information Interfaces and Presentation**]: Hypertext / Hypermedia – *Architectures.*

General Terms

Design, Languages, Standardization.

Keywords

Presentation integration, presentation composition, presentation component, component model, user interface (UI), XPIL.

1. INTRODUCTION

Creating composite applications from reusable components or modules is an important technique in software engineering and data management. A large body of research and development exists in integration-related areas such as enterprise application integration (EAI), enterprise information integration (EII), and service composition. However, most of these efforts focus on

simplifying integration at the data or application level, while little work has been done to facilitate integration at the *presentation* level. It is well-recognized that the development of user interfaces (UIs) is one of the most time-consuming parts of application development [8], so this indicates that reuse is also critical at the presentation level. However, UI development today is mostly facilitated by toolkits (e.g. Java Swing) providing pre-packaged classes modeling fine-grained UI controls such as buttons and menus; the integration of high-level presentation components encapsulating reusable application functionalities has received little attention.

The need for integrating coarse-grained components at the presentation level is manifest and examples are numerous, both in the enterprise and the consumer space. Indeed, hundreds of examples of presentation integration exist today, in the form of web mashups [7] (see ProgrammableWeb.com for a list of popular mashups). Web mashups perform integrations both at the application level and at the presentation level. However, since there is very little support in terms of model and tools for presentation integration, the presentation aspect of most mashups today is developed manually. That is, a developer needs to glue the UI of the desired components together using scripts or general purpose programming languages, in an ad-hoc fashion. Most of the developer's time is spent in trying to figure out the programming interfaces of the components, and then use the appropriate runtime and languages to integrate them.

This situation is similar to that witnessed at the dawn of data and application integration, where the need for integration was present but methodologies and tools were not. People resorted to hacking components and information together by writing all the integration logic from scratch, using conventional programming languages such as C or SQL. Eventually, the importance of reuse and of structured approaches to integration supported by tools was recognized, and entire multi-billion dollar industries came to life in the space of EII and EAI. We argue that a similar path will need to be followed by presentation integration.

Following our preliminary investigation [3], in this paper we introduce a framework for integration at the presentation level; that is, integration of components by combining their presentation front-ends, rather than their application logic or data. The granularity of components is that of stand-alone modules or applications encapsulating reusable functionalities; the goal is to build composite applications that leverage the components' individual UIs to produce composite applications possibly with rich and highly interactive user interfaces.

The framework builds on lessons learned in data and application integration but extends and adapts them to the specific needs of the presentation layer. Specifically, we argue for the need of the notion of *presentation component*, a loosely-coupled, coarse-
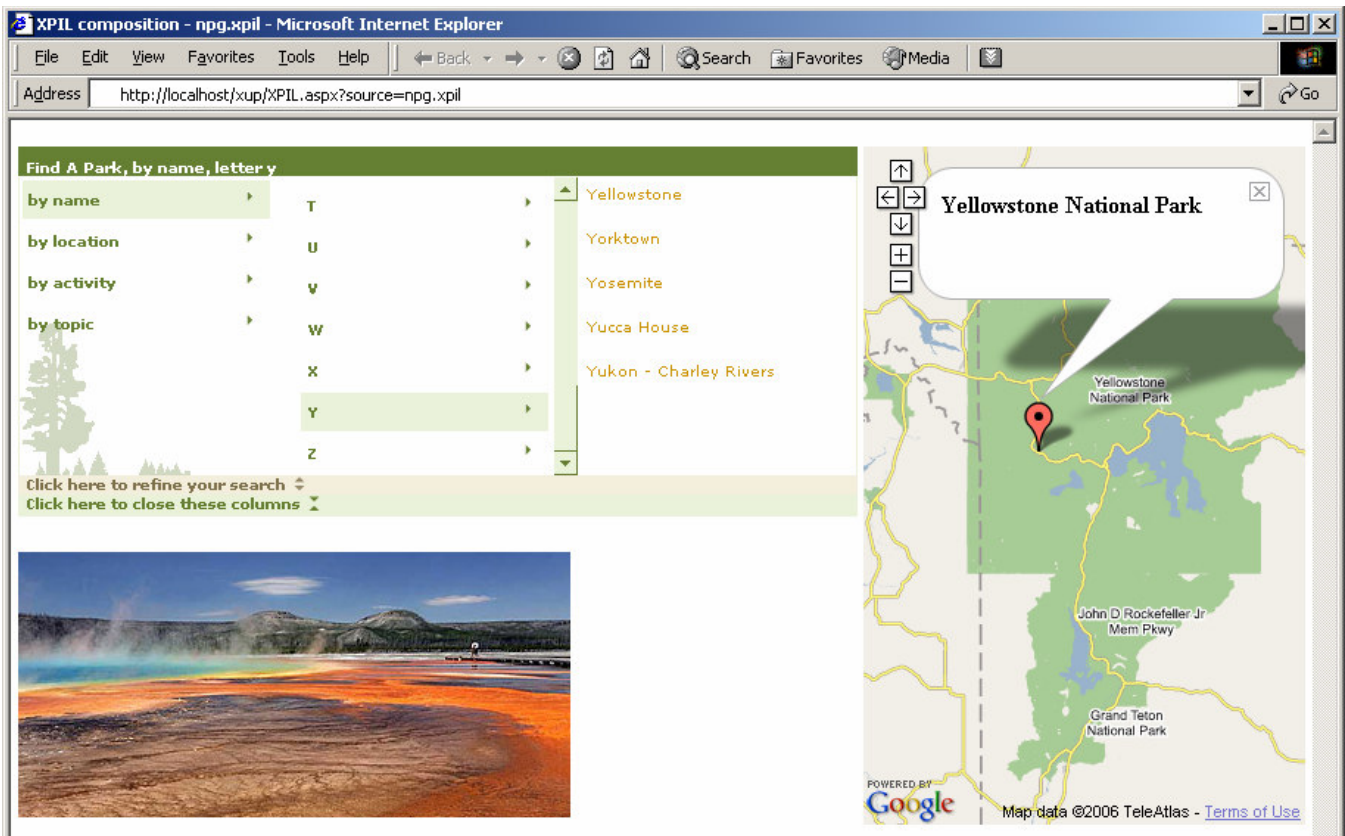
Figure 1. The National Park Guide.

grained module or application which includes a presentation layer (i.e. UI and logic to manage user interactions), that offers a programmatic access to facilitate its integration with other presentation components into an overall user interface. We also argue on the need for a composition framework (model, language, and tools) that allows the development of composite applications from presentation components, and of a runtime infrastructure that manages the interactions among components and keeps them synchronized with respect to the content they are displaying.

The end goal is that of being able to drag and drop components on a canvas and quickly specify the UI integration logic so that a complex application can be built by aggregating components with minimal development effort. These presentation components should also be easily reusable in various composite applications and, conversely, a composite application would ideally be able to swap between components providing similar UI functionality (e.g., different map providers or different image feed providers).

1.1 Reference Scenarios

To understand the problem and the need for such a framework, consider the development of a US national park interactive guide (see Figure 1). There are three presentation components in this example: a national park listing which contains a list of US national parks, an image displayer which shows images given a point of interest, and a map which displays the location of a given address or point of interest. When the user selects a national park from the park listing component, the image displayer will show an image of the selected park while the map will display its location.

Instead of building the above three presentation components from scratch, we choose to reuse existing components. For the national

park listing component, we can leverage the "Find a Park" service from the web site www.nps.gov. For the image displayer, we can use the Flickr.NET component, which displays images given some keyword tags. And for the map service, we can use Google Maps, which displays the location map given a point of interest.

For the above example, one can manually build a composite application using client-side JavaScript to maintain the coordination among the components, so that the selection of a park name causes the map and the image to change. Most of web-based presentation integrations are done with this approach, which has several important drawbacks: the developer needs to be intimately familiar with the details of each component, the integration code is not reusable, and components become tightly coupled. In fact, if developers want to switch components (e.g., use MapQuest instead of Google Maps) or "reuse" Google Maps and Flickr in other applications, the development effort is significant.

Another very common example is the integration of UIs within enterprise applications. For example, there are companies such as HP offering consoles for IT management, service management, and process management, separately developed over time or through acquisition. Ideally, users want a single enterprise console that integrates these more specific consoles to have an overall view of a business process, of the services supporting this process, and of the IT infrastructure supporting the services. Note that integration does not just mean to put the three GUIs side by side: interactions need to be coordinated so that for example user interactions with one component UI (e.g., visualization of a process) affect what is displayed by the other UIs (e.g., displaying

information on services and the IT infrastructure used by that process).

1.2 Contributions to Web Engineering

In light of the previous considerations, we believe that the potentials for a presentation integration framework cannot be emphasized enough, and that there are huge opportunities for research and development in this area. In this paper we aim at laying the foundations for such a framework and at providing a proof of concept implementation. Specifically, we make the following contributions:

- We present a model for presentation components, aiming at combining simplicity with effectiveness. The key observations are that presentation components require i) a conceptual, application specific notion of state (e.g., the location and the zoom level for maps, the service or process for enterprise management applications), ii) operations to request state changes, iii) events to notify state changes, mainly occurring due to user interactions, and iv) layout and appearance characteristics to give a consistent look and feel to the composite application.

- We propose an event-based composition model and a corresponding lightweight middleware, as we argue that presentation integration is mostly event-based. For cases when event-based specification is insufficient, additional integration logics may also be specified in the form of simple scripts or references to external code.

- We provide bindings from the abstract component model to concrete component implementations, leveraging an adapter framework for communicating with existing heterogeneous presentation components.

In the next section we discuss some background concepts, especially with respect to application integration. Section 3 describes the proposed presentation integration framework. We then illustrate a detailed example in Section 4, followed by a brief discussion of implementation issues in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2. GUIDING PRINCIPLES

In this section we discuss the characteristics of the presentation integration problem, in particular in terms of similarities and differences with respect to application integration.

2.1 Lessons Learned from EAI

A plethora of research is available in the fields of integration. Although integration problems and solutions differ based on the kind of integration needed, certain issues appear to be common and certain approaches seem to be more successful and applicable than others. A key learning from research in EAI is the need for a homogeneous way to describe the different components to be integrated. This description should be *simple, formal, human readable*, and *modular*.

Simplicity is paramount: it has been proven over and over that complex models and languages do not succeed. In application integration, only simple languages made it into the mainstream use, such as IDL and WSDL. Formalization is needed as the tool support is essential. Tools relevant for integration include both development environment as well as runtime middleware that handle binding and interaction. Readability is important as, although tools often act as mediation between a language representation and the user, developers often need to read the specifications directly (e.g. to overcome inflexibility of the tools).

Modularization is essential to disseminate a new integration model. Approaches that tried to push a single specification to cover all aspects in a *big bang* approach had very limited success. The problem here is that, first, the learning curve should be small and developers only want to learn what is needed for the case they are handling; second, and most importantly, the requirements become clear only after a technology is being used. Hence, the best approach is to start simple, understand requirements, and then add additional functionalities later if needed. This is for example the path adopted by Web services, which started with a very simple model, language, and protocol (SOAP and WSDL) and then added additional features over time (coordination, transaction, reliability, etc.), and is contrary to the path followed by ebXML, which had a much lesser success.

Another interesting lesson, borrowed from application integration, is the success of queue-based, publish/subscribe, and bus-mediated approaches to interoperability [2]. This has been proven by the success of EAI and message broker platforms, and by the fact that even in Web services, originally born for fully decentralized interaction with no assumption on a common middleware, the notion of enterprise service bus quickly emerged and now it is the common approach to implement SOAs, at least within the enterprise.

Finally, we observe that there is no easy solution to syntactical and semantic heterogeneity in application integration. In the end, the solutions adopted amounts to allowing the specification of mapping and transformation so that data can be exchanged among components, possibly with the aid of tools that facilitate data matching and mapping definitions [2].

2.2 Differences between Presentation and Application Integration

The above observations provide us with general principles and guidelines to face the problem of presentation-level integration (PI). There are, however, important differences that we need to keep in mind when developing an integration framework at the presentation layer.

A major difference is that PI is typically *event-driven*, and specifically driven by end users' actions. When the user interacts with the UI of a component, it will react according to its own UI behavior which may result in certain state changes. At this point, the rest of the components in the same composite application need to be aware of the UI state changes in the first component, so that they can update their UI accordingly.
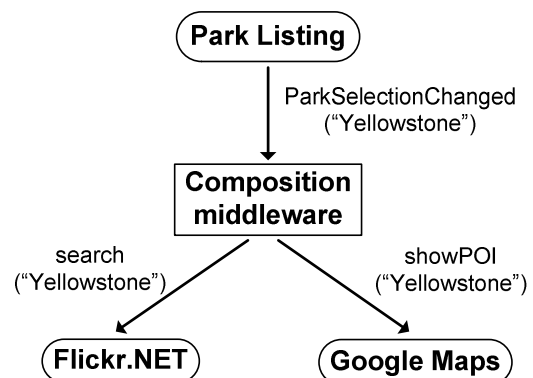

Figure 2. National park guide (event-based model).

In our national park example, this means that when the user selects a different park from the park listing component, this component would fire a "ParkSelectionChanged" event (Figure 2).

This event notifies Flickr and Google Maps to update their UI accordingly (i.e. displaying the image and the map of the newly selected park). Loose coupling here advices the use of an intermediation as opposed to implementing point to point links among components. As we will see this loose coupling is achieved via an event broker.

Hence, communication among components mainly consists of notifications of (and requests for) state changes. This means that, intuitively, we need a notion of application-defined *state*, whose data type is also application specific. In a composite application, what is important for the purpose of UI coordination is being able to manipulate a component's state as well as to detect its state changes.

This is unlike EAI where a component offers an arbitrary set of methods consisting of invocation and reply data, possibly complex and/or with large attachments. Furthermore, in EAI, the integration is mainly procedural, achieved via the specification of fairly complex control logic (e.g., in BPEL [11] or other workflow-like language) that causes the invocation of services, typically in some predefined sequence. The interaction with the individual component is fairly complex as well and possibly regulated by a business protocol. EAI components also typically do not have a first class, application-specific notion of state.

Another difference is that presentation components often require the configuration of UI appearances, such as font and background color. Hence, we need a notion of *configuration parameters*, for the purpose of design-time component customization. For example, a developer can specify the font and background color of a map component using a visual composition tool at design time. This is not commonly used in EAI, where the notion of configuring a service before using it is rare and not part of the mainstream component models or description languages.

In presentation integration the runtime middleware needs to know if the UI is visible or hidden, minimized or maximized; that is, the middleware should be able to monitor, query, and update the *presentation modes* of the components. In addition, components in PI also require proper layout management; this includes, for example, the location, size, shape, transparency, and z-order of the presentation components.

Finally, EAI is characterized by hard requirements in terms of reliability, transactionality, and security. In the typical applications of PI this level of reliability and security is not expected to be of crucial importance, meaning that the extra complexity generated by reliability and security requirements may not be justified. Hence, at least in the initial proposal for a PI solution, and until if and when such requirements materialize, we will not put emphasis on reliability and security.

3. PRESENTATION INTEGRATION FRAMEWORK

Based on the previous considerations and requirements, we propose in this section a conceptual model as well as a framework to facilitate presentation integration. Figure 3 describes the high-level architecture of the proposed framework for the execution of composite applications.

A composite application consists of one or more *components*, a specification of the *composition model* (i.e. integration logics that coordinate the components at runtime) and a *middleware* for the execution of the composition. The middleware includes an event broker that manages a set of *event listeners* defined in the composition model. The event listeners map state change *events,*

generated by one component, onto *operations* (i.e. state change requests) of other components.

The specification of the composition is performed by the application *composer* (i.e. composition developer) at design time, who may also consult a proper component *registry* to identify presentation components that suit his/her application requirements by inspecting the respective abstract *component descriptors*. Component descriptors are similar to WSDL descriptors of Web services; however, as we will show in the following, some characteristic differences apply in the case of presentation components.
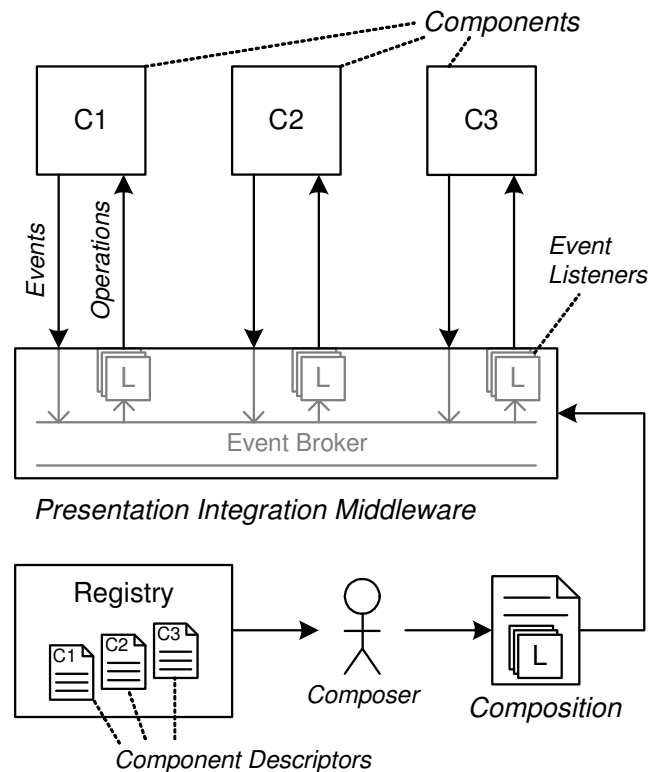

Figure 3. Architecture of the proposed presentation integration framework.

In the following subsections we discuss the main elements of the outlined framework, namely components, composition and execution middleware.

3.1 Component Model

We propose an abstract model for presentation components, where *abstract* means that it is not tied to specific implementation technologies, and that it should be able to describe existing presentation components from heterogeneous component technologies.

Conceptually, a component is characterized by a *state*, which defines what the composite application can see and control in terms of changes to the UI. The state can be complex and consist of multiple attributes (e.g., map location and zoom level). A set of *events* allow notification of state changes, while *operations* allow for querying and modifications of the state.

In addition, presentation components typically have *configuration parameters* that reflect UI appearances such as font face and background color. Parameters are specified at design time (or component creation time) and can no longer be modified at

runtime. Configuration parameters are therefore exposed via a set of *properties*, allowing the inspection and specification of the parameter values at design time.

In general, the attributes of the component's state are high level and conceptual (e.g., location and zoom level), while configuration parameters are related to preset graphical attributes (font faces, background colors, etc). However it is up to the component developer to define what characteristics are part of the state and what characteristics are configuration parameters. Ideally, the state should be kept as simple as possible to facilitate integration and reuse, as state changes are what cause events to be exchanged among components and therefore need to be handled in the composite.

The external interface (i.e. the component model) of a presentation component consists of a set of events, operations, and properties, which allow the component to expose its state and configuration parameters. To better illustrate the concepts, we will use the following XML fragment, which contains a list of component model descriptors (<component> elements) that correspond to the park listing, Flickr, and Google Maps, respectively.[1]

```
<component id="parkListing"
  xmlns:cm="http://www.openxup.org/2006/xpil/component"
  adapter="org.openxup.adapter.SackAdapter"
  address="http://www.nps.gov/findapark/index.htm">

  <event name="ParkSelectionChanged"
      address="selectPark">
    <param element="nps:parkName"/>
  </event>
</component>

<component id="imageDisplayer"
  xmlns:cm="http://openxup.org/2006/08/xpil/component"
  adapter="org.openxup.adapter.dotNETCompAdapter"
  address="http://.../FlickrNet.dll">

  <operation name="search" address="PhotosSearch">
    <input element="nps:tags"/>
  </operation>
</component>

<component id="map"
  xmlns:cm="http://openxup.org/2006/08/xpil/component"
  adapter="org.openxup.adapter.GMapWrapper"
  address="http://maps.google.com/maps?file=api...">

  <operation name="showPOI" address="showAddress">
    <input element="nps:POI"/>
  </operation>

  <property name="currentLocation">...</property>
</component>

<types
  xmlns:cm="http://openxup.org/2006/08/xpil/component">
  <!-- data types defined by XML Schema, for
  events, operations, and properties -->
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/..."
      targetNamespace="http://nps.gov/2006/...">
    <xsd:element name="parkName" type="xsd:string"/>
    <xsd:element name="tags" type="xsd:string"/>
    <xsd:element name="POI" type="xsd:string"/>
  </xsd:schema>
</types>
```

Listing 1. Component model descriptors.

---

[1] Note that component model descriptors may in fact come from different developers. For example, the XML fragment in Listing 1 could be created by three different developers, each providing the component model for one of the components.

Now we will proceed with the details of the component's external interface.

Events. A presentation component may expose a set of events to notify other components of its state changes, which are initiated either by user actions on the UI, or by requests from other components. For example, the park listing component will fire a "ParkSelectionChanged" event when the user selects a different park (see Listing 1).

Note that our component model is only concerned with component-defined events, not native UI events defined by the underlying UI toolkit. Figure 4 illustrates the distinction between component-defined events and native UI events.
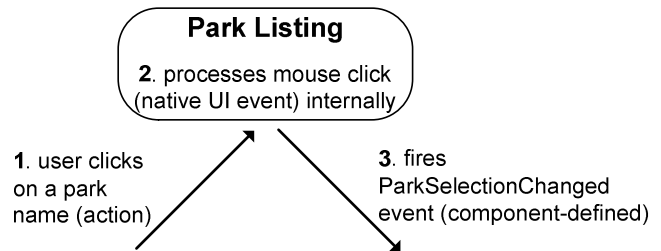
Figure 4. Component-defined event vs. native UI event.

Essentially, user actions trigger both native UI events and component-defined events. However, native UI events are captured by the underlying UI toolkit and processed by the components internally, whereas component-defined events (which signal state changes) are exposed externally. It is up to the component to define and implement the relationship between native UI events and component events that signal state changes.

Operations. A presentation component can expose a set of operations that allows for queries and modifications of its state. In our national park example, the map component supports an operation called "showPOI" (see Listing 1), which displays the map given a point of interest. An operation typically supports a list of input parameters which allows the caller to pass in values, and a return value which allows the caller to retrieve the result. The support of multiple input values allows an operation to set an attribute of the component state with various options, or even to set multiple attributes of the state at the same time (e.g. setting map location and zoom level within a single operation).

Properties. At design time or component creation time, properties can be used to expose the initial state and the configuration parameters of the component. For example, properties allow the design-time customization of the map component's configuration parameters such as font face and background color, and initial state such as the default map location.

At runtime, properties can be also used to expose component's state (e.g. the "currentLocation" property of the map component in Listing 1, which allows for the query or update of the current map location at runtime). However, unlike operations, a property is usually expressed as a pair of setter and getter, supporting a single value. That means that properties are simpler and easier to manage than operations, and therefore more suitable for visual composition tools at design time or deployment time.

Presentation modes. In addition to events, operations, and properties, there are characteristics common to all components which allow the runtime middleware to properly manage the component's execution. Collectively, we call them presentation modes, which include:

- Component's visual appearance characteristics, such as its *visibility* (visible or hidden) and *window state* (minimized or maximized);
- Component's lifecycle information. A component can be in one of the following *lifecycle states*: *instantiated* (downloaded and instance created), *ready* (finished initial configuration and ready to handle tasks), *busy* (busy processing tasks), and *destroyed* (instance destroyed).

Presentation modes are different from component properties: their semantics must be understood by the runtime middleware for the components to be properly managed. As a result, the runtime middleware should be able to monitor, query, and update the presentation modes of a component.

3.2 Composition Model

The composition model includes event subscription information to facilitate the communication among presentation components. In addition, the composition model may contain additional data transformation logics via XSLT [18] and integration logics in the form scripts or references to external code. Finally, the composition model also includes layout information so that the presentation components can be positioned properly.

Again, we will use our national park example to better explain the concept. The following XML fragment describes the composition model of the example.[2]

```
<listener id="parkChangedImgListener"
  xmlns="http://www.openxup.org/2006/08/xpil/integration"
  publisher="parkListing"
  event="ParkSelectionChanged"
  subscriber="imageDisplayer"
  operation="search"/>

<listener id="parkChangedMapListener"
  xmlns="http://www.openxup.org/2006/08/xpil/integration"
  publisher="parkListing"
  event="ParkSelectionChanged"
  subscriber="map"
  operation="showPOI"/>

<layout manager="CSS2" xmlns="http:.../xpil/integration">
  ...
</layout>
```

Listing 2. Composition model description.

Event subscriptions. Components exchange events through an event broker that facilitates loose coupling. The composition model supports a one to many publisher/subscriber relationship among presentation components. That is, one component publishes an event (i.e. declares that it will fire an event), and other components subscribe to it (i.e. declares that they will listen to and handle this event). In our national park example, the image displayer and the map component (subscribers) listen to the park selection changed event from the park listing component (publisher).

The publisher/subscriber relationship is specified via *event listeners*. Each listener specifies an event publisher, event type, event subscriber, and an operation of the subscribing component. In addition, multiple event listeners can be used to support

---

[2] Note that Listing 2 contains references to the presentation components defined earlier in Listing 1. In general, component model descriptors are first created by one or more component developers; then the composition developer authors the composition model by referencing the components defined in the component model descriptors.

multiple event subscribers for a single event from the event publisher. Note that to facilitate loose coupling, event listeners are specified in the composition model, not in the component model descriptors of the subscribing components.

Our national park example (Listing 2) contains two event listeners: one links the "ParkSelectionChanged" event from the park listing component to the "search" operation of Flickr, and the other links the "ParkSelectionChanged" event from the park listing component to the "showPOI" operation of Google Maps.

Data mappings. When direct mappings between event parameters and operation parameters are impossible, additional mappings and transformations can be specified inside event listeners. Specifically, inline or external XSLT style sheets may be specified in the event listeners to define data transformation logics for mapping the event parameters to operation parameters.

Additional integration logic. The primary goal of the composition model is to facilitate the declarative composition of presentation components. However, additional integration logic may be needed (e.g. via simple scripting languages) for those infrequent occurrences when the integration cannot be entirely declared in the composition model. For example, a location change on a map may be expressed in terms of (latitude, longitude) coordinates, and there may be the need to invoke an external service to derive city or state information from such coordinates, and then update Flickr topics with such information. In addition, a composite application may need finer control of the integration process, through the direct invocations of operations and properties of the presentation components. That is, a developer can build a composite application by writing code on top of the declarative composition framework that directly calls the operations and properties of individual presentation components. This allows the developer to directly manipulate the state of the presentation components and pass data among them.

Therefore, the composition model allows additional integration logics to be specified within event listeners, in the form of simple inline scripts or references to external code. The supported scripting or general purpose languages depend on the middleware implementation. Our current prototype supports JavaScript, either embedded inline or as external files. The reason behind this is that we believe that the exact requirements for an abstract scripting language will become clear as experience is gained with presentation integration. At this stage, JavaScript suits our purpose.

Layout information. The composition model itself does not define any layout mechanism, but supports the notion of external layout managers. This design facilitates maximum reuse of existing layout technologies while at the same time providing a flexible and extensible layout service for presentation integration.

Layout information may be specified in a <layout> element (see Listing 2). The content of this element is not interpreted by the middleware; instead it is simply passed to the external layout manager at runtime. In addition, presentation components typically expose layout properties, such as x, y, width, and height (i.e. as part of the component model). At runtime, the middleware will pass these properties to the external layout manager. When combined with the layout specification in the <layout> element, these properties allow the external layout manager to properly position the presentation components at runtime.

3.3 Language Representation

To facilitate the easy integration of presentation components, we propose a declarative composition language, the Extensible

Presentation Integration Language (XPIL). The language contains two sets of XML elements, one for describing the component model, and the other for describing the composition model.

The component model consists of a list of component descriptors (<component> elements) and XML Schema type definitions (<types> element), and the composition model contains a list of event listeners (<listener> elements) and layout information (<layout> element). Listing 1 shows an example of component model description, and Listing 2 shows an example of composition model description.

The component and composition models are typically created by different developers, and they are usually authored in multiple files (e.g. one file for the composition model, and one file for each component model). To make the distinction clear, we made the XML elements describing the component model and the ones that describing the composition model under different XML namespaces. This provides a clear separation between the two models, even if they are authored in the same document.

In designing XPIL, we try to leverage existing standards from application integration. As shown in Listing 1, the <operation>, <input>, and <types> elements are very similar to the corresponding ones in WSDL 2.0. In addition, the structure of XPIL documents is also very close to WSDL documents. For simplicity and ease of authoring, XPIL currently does not require separate sections for binding and endpoints definitions. The <component> element combines similar functionalities of WSDL 2.0's interface, binding, and service elements.

3.4 Runtime Middleware

The runtime middleware integrates presentation components, by leveraging information in the composition model. There are two key ingredients in the middleware. First, the middleware offers an event automation mechanism which allows the invocation of designated component operations in response to events; second, it provides an adapter framework for connecting to components from heterogeneous component technologies.

In addition, though not discussed in this paper, the middleware also supports common services such as data transformation, component naming, location, and lifecycle management. Our middleware currently does not provide advanced features found in EAI, such as transactions and queues. As stated earlier in section 2, we want to start simple and hence, will not emphasize on non-functional aspects such as security or reliability. Following examples in service composition (e.g. WSDL), those features can be added later if and when needed.

Event automation. To facilitate the declarative specification of presentation integration, the middleware supports the notion of *event automation*. Via event automation, the middleware captures an event from a source component and automatically dispatches it to the designated operations of other components, based on the event listener specifications in the composition model.

Conceptually, this is similar to how message brokers and event buses behave, with the difference that there is no explicit subscription done by the components (i.e. in the component model). Instead, the event subscriptions are specified via event listeners in the composition model.

With the traditional subscription model, the subscriber must be aware of the event it subscribes to. Therefore, there is a tight coupling, not with the publisher but with the event (often called *topic*) being published. To avoid this tight coupling, the definition of which events cause which operations to be invoked, as well as

of the data mapping required, must reside in the composition model, not the component model. As a result, our middleware can automatically perform transformations from events raised by one component onto operations of other components.

Figure 5 provides a simple illustration of what happens at the runtime, using our national park example:
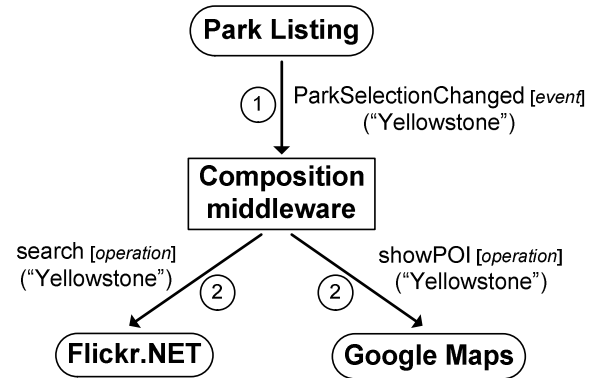

Figure 5. Event automation.

1. Capturing event from the publishing component
   a. The park listing component fires the event "ParkSelectionChanged".
   b. The middleware captures this event.
2. Automatically invoking operations of the subscribing components
   a. The middleware searches for a list of event listeners matching this event.
   b. For each listener, the middleware executes the data transformation logic (if any) that maps event parameters to operations parameters, and then invokes the specified operation on the subscribing component. In our example, the "search" operation of Flickr and the "showPOI" operation of Google Maps will be invoked.

In summary, the event automation mechanism goes one step further than the traditional event publishing and subscription mechanism: it facilitates the automatic invocation of component operations in response to events. In addition, event subscriptions are specified in the composition model, not the component model. This lays a solid foundation for the declarative composition of loosely coupled presentation components.

Component adapters and wrappers. In order to support heterogeneous components, the runtime middleware supports the notion of *component adapters*, which allow the middleware to communicate with components from different component technologies. Using these adapters, the middleware will permit the integration of presentation components developed using a wide variety of technologies, as long as the corresponding component adapters are available. For example, in our national park guide, the park listing is an AJAX component built with Simple AJAX Code-Kit (SACK) [15], Flickr is a .NET component, and Google Maps is another AJAX component.

Specifically, a component adapter performs the following functionalities:

- Component location and instantiation: locating the component implementation through URI, local class name, etc., and then creating an instance of the component.

- Component inspection: identifying the native addresses of events, operations, and properties within component implementation, through means such as reflection. This implies, for example, being able to map an event to an event member in a .NET class and map an operation to a method in a Java class, etc.

- Data type mapping: mapping the component's native data types to and from the platform-independent data types used in the component model (i.e. XML Schema types).

- Component invocation: capturing native component events and exposing them as the appropriate abstract events defined in the component model; invoking operations and properties by executing their corresponding native counterparts in the component implementation.

Through appropriate component adapters, the middleware can practically interface with any component technologies, and therefore be able to compose existing presentation components from a variety of sources. Figure 6 illustrates the adapter framework.
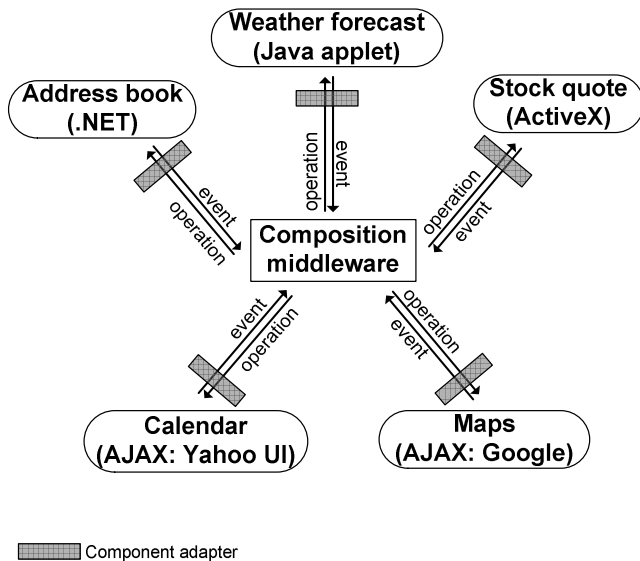
Figure 6. Component adapters.

Referring to Listing 1, the "adapter" attribute of <component> specifies the adapter to be used by the middleware to communicate with the component, and the "address" attribute specifies the location of the component which allows the adapter to download and instantiate the component. In addition, <event> and <operation> also contains an "address" attribute, which allows the adapter to identify the native event or operation in the component's implementation (e.g. a JavaScript function, a .NET method or event).

The adapter concept describe here applies to generic classes of component technologies, with the assumption that the mapping between events, operations, properties, and their native counterparts could be done through meta-language facilities such as reflection. However, if such meta-language facility is not available or there are no standard conventions for event registration and callbacks in a particular component technology, then a generic adapter for that class of components cannot be built. Instead, we need a component wrapper for each individual component. For example, there is no reflection mechanism or

standard convention to map the APIs of ad-hoc, custom-built JavaScript-based components to our abstract events and operations.

However, we expect the majority of presentation components are built with established component technologies (e.g. ActiveX, Java applet) or toolkit (e.g. Yahoo UI [19], Dojo [12]). Therefore, once a component adapter for a specific component technology or toolkit has been built, all components in that category can be integrated with our composition middleware.

4. EXAMPLE

The combination of Listing 1 and 2 provides a full description of our national park guide example. To conserve space, we only illustrate a single interaction in this example: after the user selects a different park in the park listing, Flickr will show a photo of the newly selected park and Google Maps will display a map of the park. Figure 1 shows the result of this user interaction. The upper-left corner is the park listing component (an AJAX component), and the lower-left corner is Flickr (a .NET component) which displays a photo of Yellowstone National Park. And at the right hand side Google Maps shows a map of the park.

At runtime, when the user selects "Yellowstone" from the park listing, the following happens:

1. The park listing component captures the user action, and fires a native event (i.e. JavaScript function "selectPark"). The component adapter in turn exposes it as the abstract event "ParkSelectionChanged" to the middleware.

2. The middleware tries to locate listeners matching this event. In this case it finds two listeners.

3. For the "parkChangedImgListener" listener:

   a. The middleware locates the component (Flickr) and the operation ("search") referred to by the listener. It then dispatches the event to the component by passing the name of the operation, "search", and the event parameter "parkName" with the value "Yellowstone" (an XML Schema string) to the appropriate component adapter.

   b. The component adapter translates the event parameter from XML Schema string to the appropriate native type supported by the component implementation, and locates the operation referred to by the listener within the component implementation ("PhotosSearch").

   c. The component adapter executes the native method, "PhotosSearch", passing in value for the "tags" input parameter (i.e. the name of the newly selected park). Note that the event parameter "parkName" and operation input "tags" are both XML Schema strings, so the value "Yellowstone" can be directly passed over without any transformation or conversion.

   d. Flickr updates its display to show a photo of the newly selected park, Yellowstone National Park.

4. The middleware performs similar steps to execute the listener "parkChangedMapListener".

The steps above illustrate the middleware's event automation mechanism. Essentially, a component publishes the events it fires via <event> elements in the component model; and the <listener> elements in the composition model define event subscriptions by linking the events to the designated operations in other components. This allows for rich interactions among loosely coupled, pre-built presentation components.

To illustrate how our framework simplifies composite application development, we shall go through the steps necessary to build our national park example.

First, component developers implement the components using whatever languages or technologies they prefer. In the national park example, since all three components are already available, this step can be skipped.

After that, they need to provide an abstract component model describing their components in XPIL (i.e. via <component>). However, this step usually does not require the involvement of the developers who created the original component implementations. As a matter of fact, any one who is familiar with the components' native APIs can author the corresponding abstract component models in XPIL. That means any existing, legacy presentation components could be integrated by simply providing component model descriptors in XPIL, as long as the appropriate component adapters are available.

In addition, it is not necessary to provide the full component model that describes every event, operation, and property of the component; instead, only the ones required for the composition need to be specified. For example, the three components in the national park example may support many addition events and operations. However, for this particular composition scenario, only the ones mentioned in Listing 1 need to be declared.

Once the component models are available, the composition author links the components together by adding event subscriptions (i.e. via <listener>) in the composition model. If event automation is insufficient (e.g. the need for complex data mappings beyond XSLT), additional integration logics can be specified in the <listener> element, as either inline scripts or references to external code. In our example, since the event parameter "parkName" and the operation input parameters "tags" and "POI" are all simple strings, there is no need for any addition data mapping or transformation.

Finally, the composition author provides layout specification (i.e. via <layout>) to position the three components appropriately. In our national park example, this is specified in CSS.

This completes the steps necessary to build our national park example. One can follow similar steps to create composite applications with much more sophisticated interactions and user interfaces.

5. IMPLEMENTATION AND DEPLOYMENT
The implementation of our prototype consists of a composition middleware to execute composite applications and a set of component adapters to communicate with existing presentation components. There are plenty of implementation alternatives. We chose the web-based model for our prototype and the web browser as the integration platform, since web browsers provide built-in support for many component technologies.

5.1 Middleware and Deployment
Our prototype includes a server-side code generator implemented in ASP.NET. Given one or more XPIL documents (e.g. one for composition model and one or more for component models) as the input, the code generator outputs a complete HTML page, including component definitions (e.g. HTML <object> tags) and the necessary JavaScript code that models the component interactions. The browser then renders this resulting page, instantiating the presentation components and executing the JavaScript which coordinates the interactions among the components. The generated JavaScript code manages event subscriptions and operation invocations. In addition, it also performs data transformation and conversion, when necessary.

Since the final composite application is executed in the browser, any additional integration logics in the composition model (i.e. inside <listener>) could be specified as JavaScript, which will be output by the code generator and executed by the browser at runtime. The JavaScript code in the composition model may refer to the component IDs as defined in the component model, since the generated HTML elements corresponding to those components have the same ID values.

Finally, since our delivery platform is the web browser, the prototype leverages CSS for layout management. Composition developers may specify any valid CSS fragment using the <layout> element in the composition model, which will be inserted into the output as is during code generation. The CSS fragment may refer to the component IDs as defined in the component model, since the generated HTML elements corresponding to those components have the same ID values.

5.2 Component Adapters
With browsers' built-in support for most popular components technologies (e.g. ActiveX, Java applet, Flash), component adapters are relatively easy to implement. In our national park example, we have implemented a .NET adapter for the Flickr.NET component[3]; this adapter could be used to integrate any .NET components. Similarly, for the park listing component, we implemented a SACK adapter, which will work with any AJAX components built with the SACK toolkit.

For Google Maps, we could implement a generic adapter which would work with many Google-based AJAX components. However, we chose to implement a wrapper for it instead, for two reasons. First, Google Maps is one of the most popular AJAX components, so developing a dedicated wrapper for it to expose many of its useful services should justify the investment. Second, the Google Maps API does not support point of interest or address directly; instead, one needs to translate a point of interest or address to geographic coordinates first, and then feed the coordinates to the appropriate API to display the map. We could leave the translation task to composition developers who would insert the proper scripts in the <listener> element. However, to make things easier, we implemented this translation logic as a JavaScript function (i.e. "showAddress") inside the wrapper.

Finally, component adapters (and wrappers) also support configuration options for component instantiation, through the <config> elements inside <component>. Examples of configuration information are user ID for Flickr service and API key for Google Maps. At runtime, the adapters will output the configuration options when called by the code generator.

6. RELATED WORK
There has been a large amount of research and development in the field of application integration and more recently service composition. Our work tries to leverage those existing work as much as possible. And in particular, the design of our composition language, XPIL, follows closely to that of WSDL. Existing developers who are familiar with WSDL will find XPIL to be easy to learn.

---

[3] There are many other APIs for Flickr. For example, we could also use an AJAX-based or Flash-based Flickr component here.

In addition, there are numerous application building frameworks, which allow developers to build composition GUI applications by assembling application building blocks or modules; for example, .NET Composite UI Application Block (CAB) [16] and Eclipse's Rich Client Platform (RCP) [13] for desktop applications, and Java Portlet [1], ASP.NET Web Parts [9], and WSRP [17] for web applications. However, these frameworks all require the components to be built using their specific interfaces or APIs. On the contrary, our component model provides an abstract layer on top of any existing component interfaces; and we do not require or enforce any specific APIs. Furthermore, since our component model is fairly generic, we believe it should be able to model existing presentation components developed in these frameworks (as a matter of fact, we are working on component adapters for the frameworks mentioned above).

Finally, there are several visual programming based frameworks that facilitate building composite web applications; for example, IBM ADIEU [10] and IntelligentPad [5,6]. Those frameworks provide a "pad" or "card" based metaphor, which presents users with a form-like interface for inputting data. The pad or card may contain, for example, snippet of HTML code or linkage to web service operations. However, with this pad or card based approach, user interactions are mostly form-based (i.e. one page or screen at a time), and therefore unsuitable for rich internet applications. In addition, it is unclear how this approach would work with AJAX-based components or legacy presentation components such as ActiveX controls or Java applets.

Our composition framework is event-based, and therefore it inherently provides richer user interactions. In addition, our component and composition models are very generic, and can be applied toward composing web applications as well as desktop applications.

7. CONCLUSION

Presentation integration is undoubtedly the next step that has to be taken in the integration area. In this article, we proposed a presentation integration framework to facilitate the creation of composite applications through a simple declarative composition language, XPIL. The language allows developers to specify an abstract component model for component descriptions as well as a composition model for presentation interaction logic.

The separation of component model and composition model allows the division of responsibility between component developers and composition developers. That is, component developers are responsible for creating presentation components using their favorite languages and component technologies, and then provide an abstract component model (in XPIL) for each component; composition developers are responsible for creating the composite application by specifying a composition model (in XPIL) with integration logics that link the components together.

In addition, we do not advocate a new interface standard for presentation components to adhere. Our proposed component model can be used analogously to WSDL at the application layer, that is, as a way to expose presentation components for the sake of integration. Indeed, when designing the language, we tried to follow existing standards in application integration, such as WSDL and BPEL. This allowed us to leverage prior work in application integration and to provide familiarity to developers who are versed in the application integration and service composition area.

Finally, for our current prototype, we chose web applications as the target of composition, since our event-based composition model is particularly well-suited for delivering rich internet applications. In addition, we chose the web browser as the integration platform due to the fact that it has broad support for various component technologies. This reduces the implementation efforts required for component adapters and wrappers.

Many improvements could be made to our integration framework. For example, the layout mechanism in our prototype is based on passing CSS fragments to the browser. We are investigating how to adapt to different layout controllers to offer more layout options. In addition, we plan to provide additional component adapters for AJAX-based toolkits, such as Yahoo UI and Dojo. This will allow a wider range of mashup applications to be developed using our framework. Finally, we are also investigating how to leverage component registries (e.g. UDDI) for component selection, and a visual authoring tool that allows the composition model to be specified in a drag-n-drop fashion with the final output generated in XPIL.

8. REFERENCES

[1] Abdelnur, A. and Hepper, S. Java Portlet Specification. <jcp.org/en/jsr/detail?id=168>

[2] Alonso, G., et al. *Web Services: Concepts, Architectures, and Applications*. Springer, 2004.

[3] Daniel, F., et al. Understanding UI Integration: A survey of problems, technologies, and opportunities. Technical Report # DIT-06-064, University of Trento, Italy. Oct. 2006.

[4] Fjellheim, T., et al. A Process-based Methodology for Designing Event-based Mobile Composite Applications. *Data and Knowledge Engineering*, Elsevier Science Publications (In Press).

[5] Fujima, J., et al. Clip, Connect, Clone: Combining Application Elements to Build Custom Interfaces for Information Access. *UIST'04*.

[6] Ito, K. and Tanaka, Y. A visual environment for dynamic web application composition. *HT'03*.

[7] Merrill, D. Mashups: The new breed of Web app. <ibm.com/developerworks/library/x-mashups.html>

[8] Myers, B. A. and Rosson, M. B. Survey on user interface programming. *SIGCHI'92*.

[9] ASP.NET 2.0 Web Parts. <msdn2.microsoft.com/en-us/library/e0s9t4ck(vs.80).aspx>

[10] ADIEU. <www.alphaworks.ibm.com/tech/adieu>

[11] BPEL4WS. <ibm.com/developerworks/library/ws-bpel/>

[12] Dojo. <dojotoolkit.org>

[13] Eclipse Rich Client Platform. <wiki.eclipse.org/index.php/Rich_Client_Platform>

[14] Google Maps API. <www.google.com/apis/maps/>

[15] Simple AJAX Code-Kit (SACK). <www.twilightuniverse.com/projects/sack/>

[16] Smart Client - Composite UI Application Block. <msdn.microsoft.com/library/en-us/dnpag2/html/cab.asp>

[17] Web Services for Remote Portlets. <www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp>

[18] XSLT. <www.w3.org/TR/xslt>

[19] Yahoo! UI Library. <developer.yahoo.com/yui/>