Orchestrated User Interface Mashups Using W3C Widgets

Scott Wilson[1], Florian Daniel[2], Uwe Jugel[3] and Stefano Soi[2]

[1] University of Bolton, United Kingdom
scott.bradley.wilson@gmail.com

[2] University of Trento, Povo (TN), Italy
{daniel,soi}@disi.unitn.it

[3] SAP AG, SAP Research Dresden, Germany
uwe.jugel@sap.com

Abstract. One of the key innovations introduced by web mashups into the integration landscape (basically focusing on data and application integration) is *integration at the UI layer*. Yet, despite several years of mashup research, no commonly agreed on component technology for UIs has emerged so far. We believe *W3C's widgets* are a good starting point for componentizing UIs and a good candidate for reaching such an agreement. Recognizing, however, their shortcomings in terms of inter-widget communication – a crucial ingredient in the development of interactive mashups – in this paper we (i) first discuss the *nature of UI mashups* and then (ii) propose an *extension of the widget model* that aims at supporting a variety of inter-widget communication patterns.

Keywords. UI Mashups, W3C widgets, Inter-widget communication

1 Introduction

If we analyze the state of the art in mashups today, we recognize that basically two different approaches have reached the necessary critical mass to survive: data mashups and UI (user interface) mashups. ***Data mashups*** particularly focus on the integration and processing of data sources from the Web, e.g., in the form of RSS or Atom feeds, XML files, or other simple data formats; mashup platforms like Yahoo! Pipes (http://pipes.yahoo.com/pipes/), JackBe Presto (http://www.jackbe.com/), or IBM's Damia [1] are examples of online tools that aim at facilitating data mashup development. ***UI mashups***, instead, rather focus on the integration of pieces of user interfaces sourced from the Web, e.g., in the form of Ajax APIs or HTML markup scrapped from other web sites; Intel Mash Maker [2] or mashArt [3] both support the integration of UI components, but most of the times these mashups are still coded by hand (e.g., essentially all of the mashups on programmableweb.com are of this type).

The mashup platforms focusing on data mashups typically come with very similar features in terms of supported data sources, operators, filters, and the like. RSS, Atom, or CSV are well-known and commonly accepted data formats, and there are not many different ways to process them. Unfortunately, this is not what happens in the context of UI mashups. In fact, there are still many different ways to look at the

problem and, hence, each tool or programmer uses its own way of componentizing UIs (both in JavaScript inside the browser and in other languages in the web server) and of integrating them into the overall layout of the mashup. As a consequence, UI components are not compatible among mashup tools, and we are far from common concepts and approaches when it comes to UI mashups.

Given for granted that UI components are able to encapsulate and deliver pieces of UIs that can be embedded into a mashup and operated by its users, the key ingredient for UI componentization we identify is the component's ability to ***interoperate*** with its surroundings, i.e., with other UI components and the hosting mashup logic. Interoperability is needed to enable components to synchronize upon state changes, e.g., in response to user interactions or internal logics. While technically this is not a huge challenge, conceptually it is not trivial to understand which communication paradigm to adopt, which distribution logic to support, or which data format to choose, maximizing at the same time the reusability of UI components across different mashup platforms, also fostering interoperability among mashups themselves.

In this paper, we approach these challenges by leveraging on a UI componentization technology that we believe will have a major impact in the near future, i.e., W3C's Widgets [4]. This choice is motivated, firstly, by the comprehensiveness of W3C's Widgets specifications family which tries to cover models and functionalities proper of the most used widget technologies existing so far, e.g., Google gadgets, Yahoo widgets and, in particular, Open Social gadgets. Moreover, the W3C consortium is a leading actor in web standards creation and its proposal already attracted important vendors that are implementing W3C's Widget compliant tools (e.g., Apache Wookie and Rave).

Specifically, in this paper, we provide the following ***contributions***:

- We discuss three types of *mashup logics for widgets* and identify a set of requirements the widgets should satisfy, in order for them to be mashed up.
- We propose an *extension of the W3C widget model* expressed in terms of an API extension and set of expected behaviours.
- We report on our experience with the *implementation* of a UI mashup following one of the described mashup logics and the extended widget model.

Before going into the details of our proposal, in the next section we briefly summarize the logic of and technologies used in the implementation of W3C widgets. Then, in Section 3, we investigate the basic mashup types for widgets. In Section 4 we specifically look at one type of mashups and derive a set of requirements for widgets. In Section 5 we propose an according extension of the W3C widget model, also providing concrete implementation examples. Finally, in Section 6 we discuss related works, in order to conclude the paper in Section 7.

2 W3C Widgets

The World Wide Web Consortium (W3C) provides a set of specifications collectively known as the *Widget family of specifications*. A Widget is defined by W3C (http://dev.w3.org/2006/waf/widgets-land/) as "an end-user's conceptualization of an

interactive single purpose application for displaying and/or updating local data or data on the Web, packaged in a way to allow a single download and installation on a user's machine or mobile device."

Widgets are made available to users by a *widget runtime* (also known as a *widget engine*). A *widget runtime* is an application that can import a widget that has been packaged according to the *W3C Widgets: Packaging and Configuration* specification [4]; the runtime may also make available at runtime any script objects required by the widget, for example the *W3C Widget Interface* [5] (the API a widget exposes to provide access to the widget's metadata and to persistently store data) or *W3C Device APIs* [6] (client-side APIs that enable the development of widgets that interact with device services like calendar, contacts, or camera). Widget runtimes are available on mobile devices, as desktop applications, or for embedding widgets in websites.

The Packaging and Configuration specification defines the metadata terms used to describe the widget (such as name, author and description) and to enable the configuration of the widget runtime. Configuration information includes the <feature> element, which can be used by the widget author to request that the widget runtime makes additional features available when the widget is running; examples of features include JavaScript APIs, libraries, and video codecs.

Within the W3C Widget family of specifications, widgets are largely conceptualized as operating independently, communicating with the widget runtime using the Widget Interface and with the client environment using standard browser features such as the Document Object Model and related JavaScript APIs.

While a widget runtime may render multiple widgets to the user simultaneously – for example, on the Home screen of a mobile device, or as part of the layout of a portal or social networking site – there are no mechanisms specified by the W3C Widget family of specifications by which the widgets communicate with each other as members of a mashup.

3 User Interface Mashups

Given a set of widgets that comply with the W3C Widget family of specifications, the question is therefore how a mashup of widgets could look like. Considering the state of the art in which widgets do not support inter-widget communications, we define a ***basic UI mashup***, as a tuple $m = \langle L, W, VA \rangle$ with:

- $L = \langle l, V \rangle$ being the *layout* of the mashup, of which l is the layout *template* (typically the template consists of an HTML page, a set of JavaScript and image files, and one or more CSS style sheets) and $V = \{v_i\}$ is the set of *viewports* inside l that can be used for the rendering of the widgets (e.g., iframes or div elements);
- $W = \{w_j\}$ being the set of *widgets* in the mashup, where each widget is of type $w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j \rangle$ with $Pref_j$ being a set of configuration *preferences* (typically, name-value pairs); and
- $VA = \{va_k | va_k \in W \times V\}$ being the set of *widget-viewport associations* needed for placing and rendering the widgets inside the mashup.

This model focuses on the layout only and is clearly not able to represent UI mashups like most of the ones that can be found on programmableweb.com. In fact, UI mashups typically are able to synchronize their widgets or UI elements upon user interactions, a feature that is missing in mashups of type m above.

Assuming now that widgets are able to communicate, in the following subsections we define three UI mashup models that are able to deal with inter-widget communications and to support widget synchronization:

- *Orchestrated UI mashups*, where the interactions between the widgets in the mashup are defined using a central control logic;
- *Choreographed UI mashups*, where the interactions between the widgets in the mashup are not defined, but instead emerge in a distributed fashion from the internal capabilities of the widgets;
- *Hybrid UI mashups*, where the emerging behaviour of a choreographed UI mashup is modified by inhibiting individual behaviours, practically constraining the ad-hoc nature of choreographed UI mashups.

We define each of these mashup types in the following, while in the rest of this paper we will specifically focus on orchestrated UI mashups, which can be considered the basis also for the development of the other two types of UI mashups.

3.1 Orchestrated UI Mashups

We define an ***orchestrated UI mashup*** as a tuple $m^o = \langle L, W, VA, C \rangle$ with:

- L being the layout as defined before;
- $W = \{w_j | w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j, E_j, O_j \rangle\}$ being the set of widgets with $E_j = \{e_{jl} | e_{jl} = \langle name_{jl}, P_{jl} \rangle\}$ being the set of events the widget can generate, $O_j = \{o_{jm} | o_{jm} = \langle name_{jm}, P_{jm} \rangle\}$ being the set of operations supported by the widget, and P_{jl} and P_{jm}, respectively, being the sets of output and input parameters;
- $VA = \{va_k | va_k \in W \times V\}$ being the set of widget-viewport associations; and
- $C = \{c_n | c_n \in E \times O, E = \bigcup_j E_j, O = \bigcup_j O_j\}$ being the set of direct inter-widget communications, i.e., message flows between two widgets connecting an event of the source widget with an operation of the target widget.

This definition of UI mashup implies that the mashup (and, therefore, the mashup developer) knows which events are to be mapped to which operations and that it is able to propagate the respective data items on behalf of the user of the mashup. This is common practice, e.g., in web service composition languages like BPEL, and does not require the widgets to know about each other.

The strength of this model is that mashups behave as they are expected to, that is, as specified in the mashup specification. A drawback is that this central mashup logic must be specified in advance, i.e., before runtime, which require a good knowledge of the used widgets by the mashup developer.

Note that in the above definition and in the following we intentionally do not introduce complex data mappings (e.g., requiring data transformation logics) or service components (e.g., requiring to follow web servie protocols), in order to keep

the model simple and focused. We however assume each inter-widget communication c_n also contains the necessary mapping of event outputs to operation inputs.

We believe UI mashups are good candidates for end user development and that data transformations or web services are not intuitive enough to them in order to profitably use them inside a mashup. Possible complex data transformations or service composition logics can always be developed by more skilled developers and plugged in in the form of dedicated widgets.

3.2 Choreographed UI Mashups

We define a ***choreographed UI mashup*** as a tuple $m^c = \langle L, T, W, VA \rangle$ with:

- L being the layout of the mashup;
- $T = \{t_n | t_n = \langle name_n, P_n \rangle\}$ being the reference topic ontology for events and operations, i.e., the set of concepts and associated parameters P_n the widgets in the mashup can consume as input or produce as output;
- $W = \{w_j | w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j, E_j, O_j \rangle\}$ being the set of widgets with $E_j = \{e_{jl} | e_{jl} = \langle name_{jl}, Topic_{jl} \rangle\}$ being the set of events the widget can generate, $O_j = \{o_{jm} | o_{jm} = \langle name_{jm}, Topic_{jm} \rangle\}$ being the set of operations supported by the widget, and $Topic_{jl}, Topic_{jm} \subseteq T$, respectively, being the set of topics an event sends data to and an operation reacts to; and
- $VA = \{va_k | va_k \in W \times V\}$ being the of widget-viewport associations.

In contrast to orchestrated UI mashups, choreographed UI mashups do not have an explicitly defined set of mappings of operations and events. Instead, each widget is capable of sending and receiving communications and of acting on them independently. Interoperability is achieved in that each widget complies with the reference topic ontology T, which provides a reference terminology and semantics each widget is able to understand. The behaviour of a choreographed UI mashup, therefore, is not modelled centrally by the mashup developer and rather emerges in a distributed way by placing one widget after the other into the mashup. That is, only placing a widget into the mashup allows the developer to understand how it behaves in the mashups and which features it supports.

The strength of this approach is that there is no need for explicit design of interactions: a developer simply drops widgets into his mashup and they autonomously interact. One weakness is that the reference topic ontology must be "standardized" (or, at least, understood by all widgets), in order for any meaningful communication to occur. This may reduce the overall richness of communication possible to a small number of fairly primitive topics – for example, location, dates and unstructured text. Another weakness is that with no predefined "plan" of the mashup, there could be the risk of the emergent behaviour of the widgets being pathological – for example, self-reinforcing loops or hunting. This could be a serious problem where the mashup components have real-world consequences, such as SMS-sending widgets or similar.

3.3 Hybrid UI Mashups

We define a ***hybrid UI mashup*** as a tuple $m^h = \langle L, T, W, VA, C \rangle$ with:

- L being the layout of the mashup;
- $T = \{t_n | t_n = \langle name_n, P_n \rangle\}$ being the reference topic ontology;
- $W = \{w_j | w_j = \langle id_j, name_j, Pref_j, version_j, height_j, width_j, E_j, O_j \rangle\}$ being the set of widgets with $E_j = \{e_{jl} | e_{jl} = \langle name_{jl}, Topic_{jl} \rangle\}$ being the set of events the widget can generate and $O_j = \{o_{jm} | o_{jm} = \langle name_{jm}, Topic_{jm} \rangle\}$ being the set of operations supported by the widget;
- $VA = \{va_k | va_k \in W \times V\}$ being the set of widget-viewport associations; and
- $C = \{c_n | c_n \in T \times O, O = \bigcup_j O_j\}$ being a set of constraints preventing operations from reacting to the publication of an event referring to a given topic.

In hybrid UI mashups, integration is achieved in a bottom-up fashion by the widgets themselves, while there is still the possibility for the mashup developer to control the interaction logic of the overall mashup in a top-down fashion by inhibiting interactions and, hence, application features that are not necessary for the implementation of his mashup idea.

The strength of this approach is that it brings together the benefits of both orchestrated and choreographed UI mashups, that is, simplicity of development and control of the behaviour. On the downside, the overall mashup logic is buried inside two opposite composition logics: the implicit capabilities of the widgets and the explicit constraints by the developer. This may be perceived as non-intuitive by less skilled developers or end users.

4 A W3C Widget Extension for Orchestrated UI Mashups

As a first step toward supporting the above UI mashup types, in this paper we aim at enabling the development of *orchestrated UI mashups*, a task that is already not possible with the W3C widget model as is. From the definition of m^o above we can, in fact, derive a set of extension requirements for W3C widgets, without which the implementation of interactive UI mashups is not possible:

1. Widgets must be able to communicate internal state changes via *events* to the outside world, i.e., the mashup or other widgets in the mashup. That is, while the users interacts with the widget, the widget must implement an internal logic that tells the widget when it should raise an event, in order to allow other widgets in a same mashup to synchronize.
2. Widgets must be able to accept inputs via *operations*, in order to allow the outside world to enact widget-internal state changes. The enacting of an operation is the natural counterpart of an event being raised. Typically, the operation implements the necessary logic to synchronize the state of the widget (e.g., the content rendered in the widget's viewport) with the event.
3. The *data formats* for the data exchanged among widgets should be kept as

simple as possible (we propose simple name-value pairs), in order to ease inter-widget communication. Considering that synchronizing widgets based on user interactions or internal state changes typically will require only the transfer of one or two parameters [3], e.g., an object identifier upon a selection operated by the user, this assumption seems reasonable. Remember that here we do not focus on web service orchestration or data processing.

We approach each of these requirements in the following sections and show how so extended widgets can be mashed up into UI mashups.

5 A Prototype Implementation

In order to better explain our ideas, in the following we adopt a by-example approach and contextualize them in our prototype implementation, finally also showing how the extended widget model can be successfully used for the implementation of orchestrated UI mashups.

5.1 Widget configuration

The W3C Widgets: Packaging and Configuration specification supports the run-time loading of extensions using the <feature> element of the widget's config.xml file. This requires that the widget runtime environment can resolve the URI of the feature to an installed capability. For example, given the feature URI *http://example.org/rpc* a runtime may install an implementation specific to that runtime environment, or a generic one if the functionality is relatively simple. If the URI is not recognized, the runtime will reject the installation of the widget if the required attribute is set to "true", but will proceed (optionally warning the user) if it is set to "false".

However, it is also possible for a W3C Widget to load capabilities dynamically while running, using <script src> elements in the HTML start file or using lazy loading techniques to dynamically insert new <script> elements based on the current context. Therefore for an orchestration interface we have to make a decision as to which approach to take in loading the required capabilities. Each has its advantages and disadvantages.

An advantage of using <feature> loading is that it gives the runtime environment the option to use server-side capabilities or augmented functionality. For example, to load an API in the widget that then talks to a high-performance server-side messaging service. The disadvantage is that if the runtime does not support the feature, then the widget is either not able to be installed, or is installed without necessary functionality. The advantage of using HTML-based script loading is that it should work in any widget runtime environment; however it is not able to take advantage of any special capabilities of the runtime. A compromise solution is to use the <feature> declaration but to set the required attribute to "false", and provide a dynamic <script> tag loader as a fallback. This enables the widget to take advantage of native runtime implementations, but has a fallback option if none is provided. This can be implemented using a fairly simple script in the widget, as illustrated in Figure 1.

```
If (widget.intercom && typeof(widget.intercom)==function){
  // the runtime has provided the intercom API
} else {
  // load the fallback library – in this case PMRPC
  widget.intercom = loader.load("pmrpc.js");
}
```

Figure 1. Widget-internal JavaScript logic to decide whether to load a fallback library or not.

5.2 Widget interface

We enable widgets to participate in orchestrated UI mashups through the specification of a so-called ***Intercom*** interface as an extension of the W3C Widget Interface. An implementation of the Intercom object must have the following three capabilities:

- It must be able to execute *operations* on the widget;
- It must be able to raise *events*; and
- It must be able to expose *metadata* about the operations and events supported by the widget.

The implementation of the Intercom interface may be made available at runtime through the use of a <*feature*> element in the widget configuration document or as a direct extension to the W3C Widget Interface specification implemented by the widget runtime.

The Intercom does not specify any orchestration configuration, but the capabilities of the orchestration participants and an interface to access the inter-widget communication features of the Intercom implementation. Therefore, we propose to introduce an attribute `intercom` to the W3C Widget Interface (see Figure 2).

```
[NoInterfaceObject]
interface Widget {
    readonly attribute DOMString     author;
    readonly attribute DOMString     authorEmail;
    readonly attribute DOMString     authorHref;
    readonly attribute DOMString     description;
    readonly attribute DOMString     id;
    readonly attribute DOMString     name;
    readonly attribute DOMString     shortName;
    readonly attribute Storage       preferences;
    readonly attribute DOMString     version;
    readonly attribute unsigned long height;
    readonly attribute unsigned long width;
     readonly attribute Intercom      intercom;
};
```

Figure 2 Widget interface extended with intercom attribute

The Intercom interface itself is defined as described in Figure 3: Inspecting the `metadata` attribute of the Intercom interface allows the widget runtime environment to obtain the list of events and operations implemented by the widget, along with their respective output/input parameters. The two functions `raise` and `call` can then be used to generate an event and to enact an operation, respectively.

```
interface Intercom {
    void raise(in DOMString operationName, in optional DOMString param1, ... );
    void call(in DOMString operationName, in optional DOMString param1, ... );
    readonly attribute IntercomMetaData metadata;
}
interface IntercomMetaData {
    readonly attribute sequence<IntercomSignature> events;
    readonly attribute sequence<IntercomSignature> operations;
}
interface IntercomSignature {
    readonly attribute DOMString name;
    readonly attribute sequence<IntercomArgument> parameters;
}
interface IntercomArgument {
    readonly attribute DOMString name;
}
```

Figure 3. A possible Intercom interface, including access functions and metadata structures.

For instance, Figure 4 exemplifies how a widget can use its Intercom to raise the events "eventName", and how an external RPC module (e.g., the one used by the specific Intercom implementation) can use the widgets' intercoms to call operations.

```
//called from widget
this.intercom.raise("eventName", arg1, arg2);

//called from communication module
widget.intercom.call("operationName", arg1, arg2);
```

Figure 4. Using the intercom object.

With the help of the Intercom interface, an automatic composition component or a composition tool can use the `metadata` attribute of several widgets to learn about the composition capabilities that the widget supports.

To keep the Intercom interface as simple as possible, we do not support operation return types or complex parameter types.

5.3 Widget implementation and behaviour

In Figure 5 we provide a possible implementation of the Intercom interface, which makes use of the external communication infrastructure (`SOMERPC`) declared as required <feature> in the widget configuration.

```
var SOMERPC = {/* some rpc module required by this Intercom implementation */};

var Intercom = function( widget ) {
    var w = widget,
        rpcmodule = SOMERPC,
        operations = {},

        // reads the meta data from a config file, xml, etc.
        metadata = rpcmodule.getMetaData( w.name ),
        raise = function( eventName ){       //init public raiseEvent method
            var args = Array.prototype.splice.apply(arguments, 1,
                                                arguments.length-1);
```

```
            rpcmodule.raiseEvent( w, eventName, args );
        },
        call = function( opName ){
            var args = Array.prototype.splice.apply(arguments, 1,
                                                arguments.length-1);
            //call widget operation if it is in the public operations
            if(operations[opName]) {
                operations[opName].apply( w, args );
            }
        },
        i = 0;

    //setup the private operations list for faster access when 'call' is executed
    for(i = 0; i < metadata.operations.length; i += 1) {
        operations[metadata.operation[i].name] = w[metadata.operation[i]];
    }

    this.raise = raise;
    this.call = call;
    this.metadata = metadata;

    //register this intercom at the rpc module
    rpcmodule.register( this );
};
```

Figure 5. A basic implementation of the Intercom interface.

The Intercom of a widget should be initialized in the widget constructor to prevent modifications from the outside:

```
// called from the widget contructor
this.intercom = new Intercom( this );
```

After the intercom is set up, a widget can start raising events via its own Intercom, and all modules that have access to the widget or the widget's Intercom can call operations on the widget.

5.4 UI mashup implementation

Using the formalization introduced in Section 3, we are able to model a variety of mashups involving multiple widgets. The specification does not include any additional runtime aspects, such as message delivery time, message buffering, or similar technical aspects. Thereby, it is flexible enough to also accomodate mashups with more complex characteristics, such as mashups involving multiple windows or multiple origins, and it is agnostic as to whether communication is purely within the browser (e.g., using HTML 5 PostMessage) or also involving the server side.

Implementing a UI mashups can be achieved relatively simply through the use of publish-subscribe services propagating events from one widget to others. In orchestrated UI mashups of type $m^o = \langle L, W, VA, C \rangle$, it is the inter-widget communication logic C that subscribes widgets, i.e., their operations, to events. In choreographed UI mashups of type $m^c = \langle L, T, W, VA \rangle$, each widget publishes its events to the topics in T and subscribes to the topics it understands. In hybrid UI mashups $m^h = \langle L, T, W, VA, C \rangle$, the bottom-up subscriptions by the widgets can be fine-tuned via the constraints C. All this can implemented using a range of existing

mature software technologies, for example, client-side using OpenAjax Hub[1] or server-side using solutions such as Faye[2] or ActiveMQ[3].

6 Related Work

In our former work [8], we developed an approach to the componentization and intercommunication of UI components. The approach is different from the one proposed in this paper, in that it aims to wrap full-fledged web applications developed with traditional, server-side web technologies. The wrapping logic requires the presence of simple event annotations inside the application's HTML markup in order to intercept events and a descriptor for the enacting of operations on the wrapped web app. Widgets, instead, are pure client-side apps.

In the context of widgets, Sire et al. [7] proposed an idea that is similar to what we propose in this paper, also advocating the use of events and event listeners (the equivalent of our operations). The widget decides whether an event is distributed in a unicast (one receiver), multicast (multiple receivers), or broadcast (all possible receivers) fashion. This design choice, however, leads to tightly coupled widgets, in that a widget must know in advance with which other and how many widgets it will communicate, a limitation we do not have in our proposal. In fact, in our case it is the mashup logic (which, for choreographed UI mashups, may be missing) that manages the inter-widget communication, and widgets are unaware of their neighbours.

The Java Portlet Specification 2.0 [9] proposes inter-widget communication for web portals. Portlets may communicate via events, but interactions occur on the server-side, a strong limitation in a UI-intensive Web 2.0 context. So far, the adoption of this technique is relatively low, also because its limitation to the Java world.

Communicating across technical boundaries, as proposed in this paper, is required in many networked computing domains. Especially for web browsers, the communication across domains and across browser windows (including iframes) is an important issue. Therefore, the HTML 5 standard defines a messaging API [10], which is, for example, used by the "pmrpc" project [11]. This project provides a Javascript module that adds a *pmrpc* object to a running website *window* object. All scripts running inside this window may access *pmrpc* to register own operations, or make calls to other windows/frames [12].

Our investigation of these and similar RPC approaches showed that different projects use different interface syntax and mainly focus on cross-window communication. In comparison to that, our proposed interface extension does not specify any cross-domain/window aspects. A single *widget*, in our case, is similar to a *window* in these related approaches, but there can be many *widgets* in many *windows* that constitute a *mashup*. All widgets will use their *intercom* transparently. Cross-domain issues must be solved internally by the Intercom implementation, which may of course use, e.g., *pmrpc* internally for this aspect.

---

[1] http://www.openajax.org/whitepapers/Introducing%20OpenAjax%20Hub%202.0%20and%20Secure%20Mashups.php

[2] http://faye.jcoglan.com/

[3] http://activemq.apache.org/

7 Conclusion and Future Work

In this paper, we addressed a relevant issue in UI-based mashup development, i.e., the intercommunication of W3C widgets. Mashups are typically heavily UI-based, but so far no standard for how to componentize UIs and how to get them into communication has emerged. We believe W3C widgets have the potential to represent this agreement and that they will gain importance in the near future in both desktop and mobile computing environments.

The aim of our research in this context is to come up with an inter-widget communication interface and respective widget behaviours, which – thanks to our involvement in the standardization of the widget technology – we would like to propose to the W3C for standardization. This is an effort we carry on in the context of the European project Omelette (http://www.ict-omelette.eu).

In order to obtain a first feedback from the community regarding the proposed communication interface, in this paper we focused on inter-widget communication at the level of events and operations. In the future, we also aim to identify and propose a standard format for the exchange of data among widgets, e.g., based on the OData protocol or similar initiatives.

Acknowledgements: This work was supported by funds from the European Commission (project OMELETTE, contract no. 257635).

References

1. M. Altinel, P. Brown, S. Cline, R. Kartha, E. Louie, V. Markl, L. Mau, Y.-H. Ng, D. Simmen, and A. Singh. Damia: a data mashup fabric for intranet applications. *VLDB'07*, September 2007, VLDB Endowment, pp. 1370-1373.
2. R. Ennals, E. Brewer, M. Garofalakis, M. Shadle, P. Gandhi. Intel Mash Maker: join the web. *SIGMOD Rec*. 36, 4, December 2007, pp. 27-33.
3. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, November 2009, Springer, pp. 428-443.
4. W3C. Widget Packaging and Configuration. *W3C Working Draft*, March 2011, http://www.w3.org/TR/widgets/
5. W3C. The Widget Interface. *W3C Working Draft*, September 2010, http://www.w3.org/TR/widgets-apis/
6. W3C. Device APIs and Policy Working Group Charter. http://www.w3.org/2009/05/DeviceAPICharter
7. S. Sire, M. Paquier, A. Vagner, J. Bogaerts. A Messaging API for Inter-Widgets Communication. *WWW'09*, April 2009, ACM, pp. 1115-1116.
8. F. Daniel and M. Matera. Turning Web Applications into Mashup Components: Issues, Models, and Solutions. *ICWE'09*, June 2009, Springer, pp. 45-60.
9. S. Hepper. Java(TM) Portlet Specification Version 2.0. Proposed Final Draft, Rev. 29. http://jcp.org/aboutJava/communityprocess/pfd/jsr286/index.html
10. WHATWG. HTML Living Standard, Communication. WHATWG specification. Website, April 2011: http://www.whatwg.org/specs/web-apps/current-work/multipage/comms.html
11. I. Kovic and I. Zuzak. Pmrpc, HTML5 inter-window and web workers RPC and pubsub communication library. Project website, April 2011: http://code.google.com/p/pmrpc/.
12. I. Kovic and I. Zuzak. List of system that enable inter-window or web worker communication. Website, April 2011: http://code.google.com/p/pmrpc/wiki/IWCProjects.