Distributed User Interface Orchestration: On the Composition of Multi-User (Search) Applications

Florian Daniel, Stefano Soi, and Fabio Casati
University of Trento, 38123 Povo (TN), Italy
{daniel,soi,casati}@disi.unitn.it

Abstract. While mashups may integrate into a new web application data, application logic, and user interfaces sourced from the Web – a highly intricate and complex task – they typically come in the form of simple applications (e.g., composed of only one web page) for individual users. In this chapter, we introduce the idea of *distributed user interface orchestration*, a mashup-like development paradigm that, in addition to the above features, also provides support for the coordination of *multiple users* inside one shared application or process. We describe the concepts and models underlying the approach and introduce the *MarcoFlow* system, a platform for the assisted development of distributed user interface orchestrations. As a concrete development example, we show how the system can be profitably used for the development of an advanced, *collaborative search application*.

1 Introduction

After workflow management (which supports the automation of business processes and human tasks) and service orchestration (which focuses on web services at the application layer), web mashups [1] feature a significant innovation: **integration at the UI level**. Besides web services or data feeds, mashups indeed reuse pieces of UIs (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into new web pages or applications. While mashups therefore manifest the need for reuse in UI development and for suitable UI component technologies, so far they produced rather simple applications consisting of one web page only.

We argue that there is a huge spectrum of applications that demand for development approaches that are similar to those of mashups but that go far beyond single page applications and, in fact, support multiple pages, multiple actors, complex navigation structures, and – more importantly – process-based application logic or navigation flows. We call this type of applications **distributed UI orchestrations** [2], as (i) both components and the application itself may be distributed over the Web and operated by different actors, (ii) in addition to traditional web services we also integrate novel JavaScript UI components, and (iii) services and UIs are orchestrated in a homogeneous fashion.

Developing distributed UI orchestrations therefore raises the need for the coordination of individual actors and the development of a distributed user interface *and* service orchestration logic. Doing so requires:

- Understanding how to *componentize UIs and compose them*;

- Defining a logic that is able to *orchestrate both UIs and web services*;
- Providing a language and tool for *specifying distributed UI compositions*; and
- Developing a runtime environment that is able to *execute distributed UI and service compositions*.

In this chapter, we describe how the above challenges have been solved in the context of the MarcoFlow project [2] and how the resulting approach can be leveraged for the development of a distributed search computing application that requires the coordination of services, UIs, and people.

This chapter is organized as follows: Next, we describe the search application that raises the need for distributed UI orchestration. In Section 3, we look at how existing techniques and technologies may support the development of such kind of application, while in Section 4 we introduce the distributed UI orchestration approach in order to fill the gaps. In Section 5, we describe our current development prototype, and in Section 6 we conclude the paper.

2 A Search Scenario

Let us consider the following collaborative search scenario illustrated in Figure 1, an extension of the single-user scenario discussed in [3].

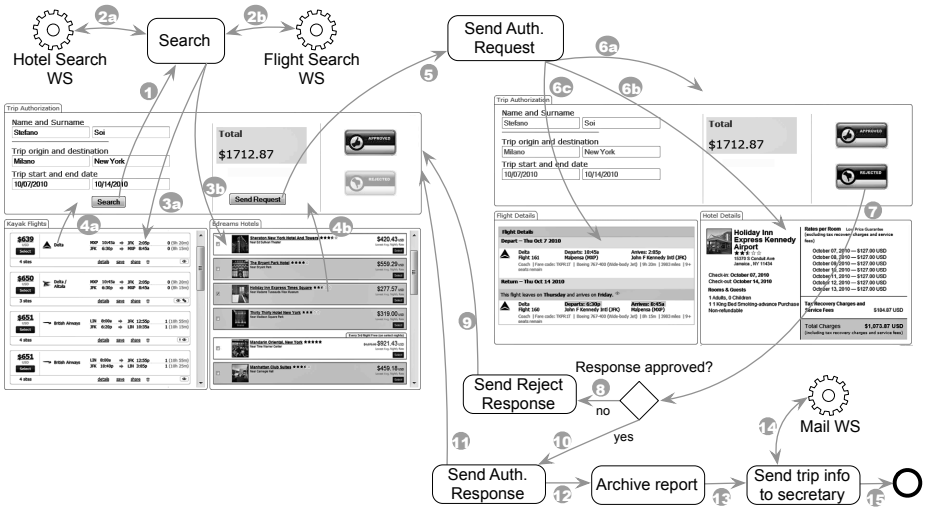

Fig. 1 Trip authorization distributed application. The gray arrows indicate synchronization or orchestration points; the number labels indicate their order in time.

This time we want to assist an employee that needs to request the authorization of a business trip to his superior. The employee can enter the relevant information about the trip (the origin and destination, and the start and end dates) and search for related flights and accommodations. He/she can select his/her preferred choices from the list of flights and hotels and send the request to the superior. The superior can inspect the

request, along with its details, and send a response (accept or redo) to the employee, together with some optional comment. If the request is approved, the response will be sent to the employee and the procedure will end with archiving and mailing operations. If the request is rejected the previous steps (all or a part of them) can be repeated until the authorization request is approved.

If we analyze the scenario, we see that the envisioned application (as a whole) is *distributed* over the Web. The application includes, besides the process logic, two mashup-like, web-based control consoles for the employee and the superior that are themselves part of the orchestration and need to interact with the underlying process logic. The UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the *Trip Authorization* component) or sourced from the Web (like the *Hotel Search* and the *Kayak Flights Search* component); service orchestrations are based on web services. In our case, the latter two UI components serve only to render content, which still needs to be queried from the Web via dedicated web services. The *Trip Authorization* component, instead, collects data from the user (via its input fields) and from the other two UI components (via suitable synchronization operations). Finally, the two applications for the employee and the superior are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

3 Distributed UI Orchestration

The **key idea** to approach the coordination of (i) UI components inside web pages, (ii) web services providing data or application logic, and (iii) individual pages (as well as the people interacting with them) is to split the coordination problem into two layers: *intra-page UI synchronization* and *distributed UI synchronization and web service orchestration*.

UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated via control flows. In [2] we show how to describe UI components (as also introduced in [4]) in terms of standard WSDL descriptors, how to bind them to JavaScript, and how to extend the standard BPEL language in order to support the two above composition layers. We call this extended language **BPEL4UI**. Fig. 2 shows the simplified meta-model of the language and details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded), omitting details of the standard BPEL language, which are reused as is by BPEL4UI. The model in Fig. 2 exclusively focuses on the composition aspects, while the events and operations of UI components are defined in their WSDL descriptors [2].

In terms of standard BPEL [5], a UI orchestration is a *process* that is composed of a set of associated *activities* (e.g., sequence, flow, if, assign, validate, or similar), *variables* (to store intermediate processing results), *message exchanges*, *correlation sets* (to correlate messages in conversations), and *fault handlers*. The services or UI components integrated by a process are declared by means of so-called *partner links*, while *partner link types* define the roles played by each of the services or UI components in the conversation and the *port types* specifying the operations and messages supported by each service or component.

Modeling UI-specific aspects requires instead introducing a set of **new constructs** that are not yet supported by BPEL. The constructs, illustrated in Fig. 2, are: *UI type* (the partner link type for UI components), *page* (the web pages over which we distribute the UI of the application), *place holder* (the name of the place holders in which we can render UI components), *UI component* (the partner link for UI components), *property* (the constructor parameters of UI components), and *actor* (the human actors we associate with web pages).

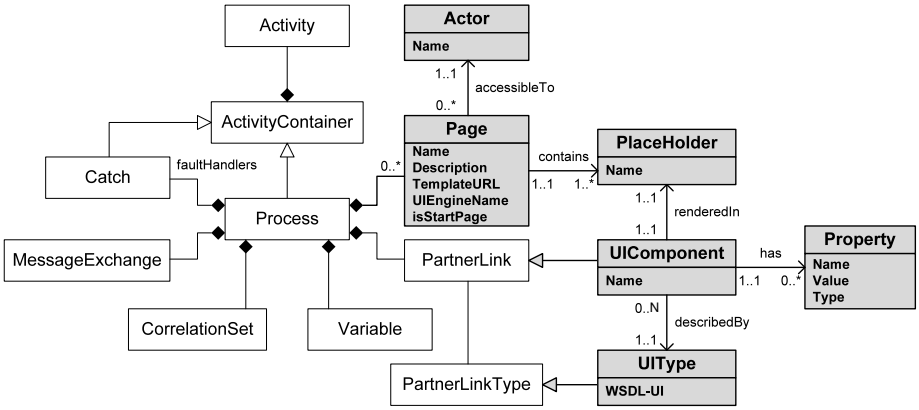

Fig. 2 Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs; gray classes correspond to constructs for UI and user management.

It is important to note that although syntactically there is no difference between web services and UI components (the new JavaScript binding introduced into WSDL to map abstract operations to concrete JavaScript functions comes into play only at runtime), it is important to distinguish between services and UI components as their *semantics* and, hence, their usage in the model will be different. A detailed description of the new constructs and their usage can be found in [2], while in Figure 3 we illustrate the BPEL4UI model of our example search application (shown in Figure Fig. 1) as modeled in our extended Eclipse BPEL editor.

The BPEL4UI model is structured into four main blocks: one *repeat-until* and three *sequences* (sub-types of the *Activity* entity in Fig. 2). The repeat-until block at the left manages the flights and hotels search operations. The processing of the block starts upon the reception of the relevant data about the trip from the employee's console, then it invokes the external search web services and, finally, sends the respective results to the UI components rendering the flight and hotel offers. This block of operations can be repeated an arbitrary number of times (e.g., in case the employee want to input new search criteria or a trip request has been rejected and needs to be redone), until the authorization is accepted. Once the search results are rendered in their UI components, the employee can choose a flight and a hotel combination by clicking on the respective choices. This allows the employee to compose his trip request summarized in the *Trip Authorization* component. The two sequence blocks (*Flight Selection* and *Hotel Selection*) in the middle of the model implement the operations that are necessary to synchronize the *Trip Authorization* UI-component, which is then in

charge of storing the combination and computing the total cost of the trip. These communications, involving only UI-components belonging to the same page, are completely managed inside the employee's web browser. Once all the trip data are available, the *Send Request* button in the employee console is activated and can be used to forward the authorization request to the superior. Receiving the authorization request starts the right block in the model (*Authorization Request and Response*), which waits for the trip request data and then forwards them to the *Trip Authorization*, *Hotel* and *Flight* UI-components of the superior's console. Now the superior can inspect the request and send a response that is forwarded to the employee's console. If the superior approves the request, two web services are invoked, respectively for archiving and mailing, and, finally, the process is terminated. If the response is a reject, the whole block of operations can be repeated, allowing the employee to modify his request. The right block of service orchestration hence requires the coordination of the two actors, i.e., employee and superior, and the distributed orchestration of UI components and web services. Doing so requires the help from the BPEL engine and the setting of a suitable BPEL correlation set.

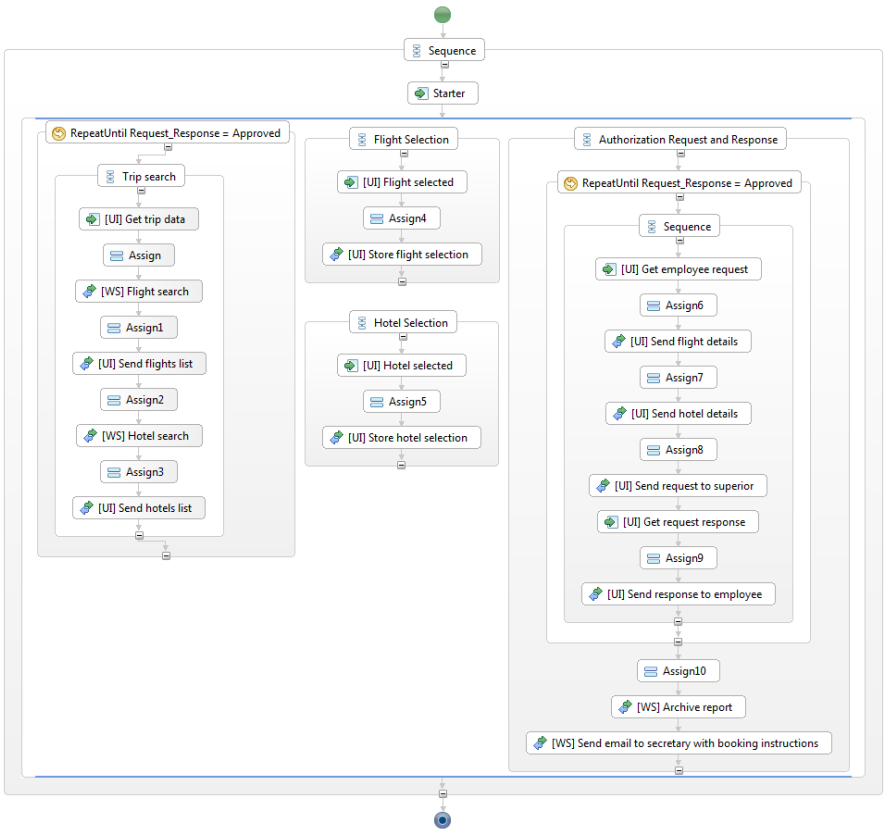

Fig. 3 BPEL4UI modeling example for the Trip Authorization application.

As for the **layout** of distributed UI orchestrations, defining web pages and associating UI partner links with placeholders requires implementing suitable HTML templates that are able to host the UI components of the orchestration at runtime. For the design of layout templates, we do not propose any new development instrument and rather allow the developer to use his/her preferred development tool (from simple text editors to model-driven design tools). The only requirement the templates must satisfy is that they provide place holders in form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention, i.e., * … *. For instance, all the activities with a "[UI]" prefix in Figure 3 are associated to placeholders, in order to fill the two pages composing our reference scenario.

As this discussion shows, the main **methodological goals** in implementing our UI orchestration approach were (i) relying as much as possible on existing *standards*, (ii) providing the developer with only *few and simple new concepts*, and (iii) implementing a runtime architecture that associates each concern to the *right level of abstraction and software tool* (e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine). These decisions, for instance, allow us to reuse BPEL's internal exception handling mechanisms to manage also exceptions in distributed UI orchestrations.

4 The MarcoFlow Environment

Fig. 4 shows the (simplified) architecture of the MarcoFlow environment, which aids the development and execution of distributed UI orchestrations. The architecture is partitioned into design time, deployment time, and runtime components, according to the three phases of the software development lifecycle supported by MarcoFlow.

The **design** part comprises the*BPEL4UI editor* that supports the full BPEL4UI language as defined in [2]. The editor is an extended Eclipse BPEL editor with (i) a panel for the specification of the pages in which UI components can be rendered and (ii) a property panel that allows the developer to configure the web pages, to set the properties of UI partner links, and to associate them to place holders in the layout.

The **deployment** of a UI orchestration requires translating the BPEL4UI specification into executable components: (i) a set of *communication channels* that mediate between the UI components in the client browser and the BPEL engine; (ii) a *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic; and (iii) a set of *UI compositions* (one for each page of the application) containing the intra-page UI synchronizations. This task is achieved by the *BPEL4UI compiler*, which also manages the deployment of the generated artifacts in the respective runtime environments.

The **execution** of a UI orchestration requires the setup and coordination of three independent runtime environments: (i) the interaction with users and intra-page UI synchronization is managed in the client browser by an *event-based JavaScript runtime framework*; (ii) a so-called *UI engine server* runs the web services implementing the communication channels; and (iii) a *standard BPEL engine* manages the distributed UI synchronization and web service orchestration.

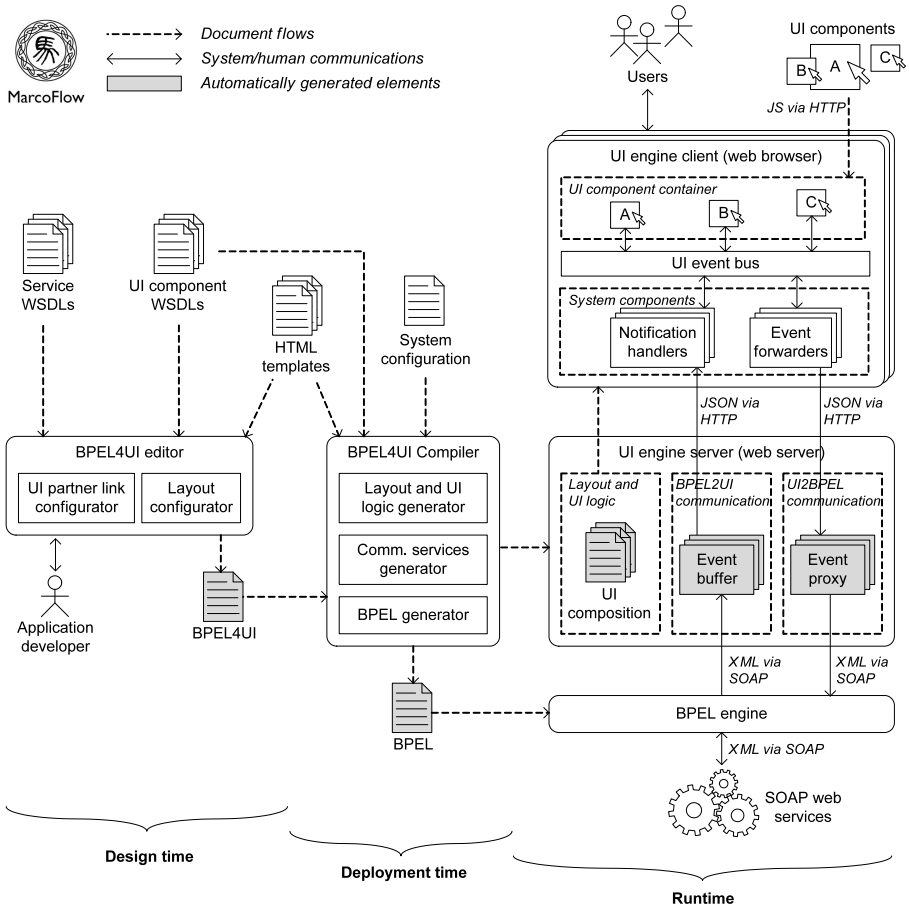

Fig. 4 From design time to runtime: overall system architecture of MarcoFlow

In order for the superior and the employee to manage their trip authorizations, MarcoFlow also comes with a simple **task manager** (not detailed in Fig. 4), which allows them to start new trip authorizations (the employee) and to participate in running instances of the application (the manager). Each new request requires a new instantiation of the process. All running instances are shown to both actors in their personalized lists. An instance terminates upon successful approval of the trip.

The MarcoFlow system shown in Fig. 4 is fully implemented and running. A patent application for parts of the system has been filed. A detailed demonstration of how MarcoFlow can be used for the development of distributed UI orchestration is available at *http://mashart.org/marcoflow/demo.htm*.

5 Related Work

In most **service orchestration** approaches, such as BPEL [5], there is no support for UI design. Many variations of BPEL have been developed, e.g., aiming at the invoca-

tion of REST services [6] or at exposing BPEL processes as REST services [7]. IBM's Sharable Code platform [8] follows a slightly different strategy in the composition of REST and SOAP services and also allows the integration of user interfaces for the Web; UIs are however not provided as components but as ad-hoc Ruby on Rails HTML templates.

BPEL4People [9] is an extension of BPEL that introduces the concept of people task as first-class citizen into the orchestration of web services. The extension is tightly coupled with the **WS-HumanTask** [10] specification, which focuses on the definition of human tasks, including their properties, behavior and operations used to manipulate them. BPEL4People supports people activities in form of inline tasks (defined in BPEL4People) or standalone human tasks accessible as web services. In order to control the life cycle of service-enabled human tasks in an interoperable manner, WS-HumanTask also comes with a suitable coordination protocol for human tasks, which is supported by BPEL4People. The two specifications focus on the coordination logic only and do not support the design of the UIs for task execution.

The systematic development of web interfaces and applications has typically been addressed by the web engineering community by means of **model-driven web design approaches**. Among the most notable and advanced model-driven web engineering tools we find, for instance, WebRatio [11] and VisualWade [12]. The former is based on a web-specific visual modeling language (WebML), the latter on an object-oriented modeling notation (OO-H). Similar, but less advanced, modeling tools are also available for web modeling languages/methods like Hera, OOHDM, and UWE. These tools provide expert web programmers with modeling abstractions and automated code generation capabilities for complex web applications based on a hyperlink-based navigation paradigm. WebML has also been extended toward web services [13] and process-based web applications [14]; reuse is however limited to web services and UIs are generated out of HTML templates for individual components.

A first approach to component-based UI development is represented by **portals and portlets** [15], which explicitly distinguish between UI components (the portlets) and composite applications (the portals). Portlets are full-fledged, pluggable Web application components that generate document markup fragments (e.g., (X)HTML) that can however only be reached through the URL of the portal page. A portal server typically allows users to customize composite pages (e.g., to rearrange or show/hide portlets) and provides single sign-on and role-based personalization, but there is no possibility to specify process flows or web service interactions (the new WSRP [16] specification only provides support for accessing remote portlets as web services). Also **JavaServer Faces** [17] feature a component model for reusable UI components and support the definition of navigation flows; the technology is however hardly reusable in non-Java based web applications, navigation flows do not support flow controls, and there is no support for service orchestration and UI distribution.

Finally, the web mashup [1] phenomenon produced a set of so-called **mashup tools**, which aim at assisting mashup development by means of easy-to-use graphical user interfaces targeted also at non-professional programmers. For instance, Yahoo! Pipes (http://pipes.yahoo.com) focuses on data integration via RSS or Atom feeds via a data-flow composition language; UI integration is not supported. Microsoft Popfly (http://www.popfly.ms; discontinued since August 2009) provided a graphical user interface for the composition of both data access applications and UI components;

service orchestration was not supported. JackBe Presto (http://www.jackbe.com) adopts a Pipes-like approach for data mashups and allows a portal-like aggregation of UI widgets (so-called mashlets) visualizing the output of such mashups; there is no synchronization of UI widgets or process logic.IBM QEDWiki (http://services.alphaworks.ibm.com/qedwiki) provides a wiki-based (collaborative) mechanism to glue together JavaScript or PHP-based widgets; service composition is not supported. Intel Mash Maker (http://mashmaker.intel.com) features a browser plug-in which interprets annotations inside web pages allowing the personalization of web pages with UI widgets; service composition is outside the scope of Mash Maker.

In the mashArt [4] project, we worked on a so-called universal integration approach for UI components and data and application logic services. MashArt comes with a simple editor and a lightweight runtime environment running in the client browser and targets skilled web users. MashArt aims at simplicity: orchestration of distributed (i.e., multi-browser) applications, multiple actors, and complex features like transactions or exception handling are outside its scope. The CRUISe project [18] has similarities with mashArt, especially regarding the componentization of UIs. Yet, is does not support the seamless integration of UI components with service orchestration, i.e., there is no support for complex process logic. CRUISe rather focuses on adaptivity and context-awareness. Finally, the ServFace project [19] aims at supporting even unskilled web users in composing web services that come with an annotated WSDL description. Annotations are used to automatically generate form-like interfaces for the services, which can be placed onto one or more web pages and used to graphically specify data flows among the form fields. The result is a simple, user-driven web service orchestration. None of these projects, however, supports the coordination of multiple different actors inside a same process, and none of the approaches discussed supports the distribution of UIs over multiple browsers.

6 Conclusion and Future Works

In this chapter, we addressed the problem of designing and orchestrating *component-based web applications* that are distributed over *multiple web browsers* and that involve *multiple different actors*. We particularly discussed the case of a search computing application that leverages on a collaborative search and browsing approach, an application feature whose development with traditional techniques would be everything but trivial. In fact, while the integration of UIs and web services is, for instance, also supported by current mashup platforms, the coordination of the actors involved in the application and the synchronization of their respective UIs would still require manual intervention. The MarcoFlow platform introduced in this chapter, instead, supports the seamless integration of services, UIs, and people in one and the same development environment, sensibly speeding up the development of process-based, mashup-like web applications.

The basic idea of MarcoFlow, i.e., the component-based development of applications is inspired by current web mashup practices, which in many cases aim at enabling also the *less skilled developer* (or even unskilled end users) to compose own applications. Given the complexity of the applications supported by MarcoFlow, it is

however important to note that MarcoFlow rather targets skilled developers (e.g., developers that are familiar with composite web service development in BPEL).

One of the challenges to be addressed in our future work is therefore lowering the complexity of the design environment for distributed UI orchestrations, hiding BPEL4UI behind an easier to learn, graphical modeling language. Also, we would like to extend the approach toward streaming web services, for example to support the design of continuous queries over sensor networks.

References

1. J. Yu, B. Benatallah, F. Casati, F. Daniel. Understanding Mashup Development and its Differences with Traditional Integration. *IEEE Internet Computing*, Vol. 12, No. 5, September-October 2008, pp. 44-52.
2. F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan. From People to Services to UI: Distributed Orchestration of User Interfaces. *Proceedings of BPM'10*, pp. 310-326.
3. F. Daniel, S. Soi, F. Casati. From Mashup Technologies to Universal Integration: Search Computing the Imperative Way. *Search Computing - Challenges and Directions*, June 2009, pp. 72-93.
4. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.
5. OASIS. Web Services Business Process Execution Language Version 2.0, April 2007. [Online]. *http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html*
6. C. Pautasso. BPEL for REST. *BPM'08*, Milano, pp. 278-293.
7. T. van Lessen, F. Leymann, R. Mietzner, J. Nitzsche, D. Schleicher. A Management Framework for WS-BPEL, *ECoWS'08*, Dublin, pp. 187-196.
8. E. M. Maximilien, A. Ranabahu, K. Gomadam. An Online Platform for Web APIs and Service Mashups, *Internet Computing*, vol. 12, no. 5, pp. 32-43, Sep. 2008.
9. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. WS-BPEL Extension for People (BPEL4People), Version 1.0. June 2007.
10. Active Endpoints, Adobe, BEA, IBM, Oracle, SAP. Web Services Human Task (WS-HumanTask), Version 1.0. June 2007.
11. R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, P. Fraternali. Web Applications Design and Development with WebML and WebRatio 5.0. *TOOLS'08*, pp. 392-411.
12. J. Gómez, A. Bia, A. Parraga. Tool Support for Model-Driven Development of Web Applications, *WISE'05*, pp. 721-730.
13. I. Manolescu, M. Brambilla, S. Ceri, S. Comai, P. Fraternali. Model-Driven Design and Deployment of Service-Enabled Web Applications. *ACM Trans. Internet Technol.*, Vol. 5, No. 3, August 2005, pp. 439-479.
14. M. Brambilla, S. Ceri, P. Fraternali, I. Manolescu. Process Modeling in Web Applications. *ACM Trans. Softw. Eng. Methodol.*, Vol. 15, No. 4, October 2006, pp. 360-409.
15. Sun Microsystems. JSR-000168 Portlet Specification, October 2003. [Online]. *http://jcp.org/aboutJava/communityprocess/final/jsr168/*
16. OASIS. Web Services for Remote Portlets, August2003. [Online]. *www.oasis-open.org/committees/wsrp*
17. Oracle. JavaServer Faces Technology. [Online] *http://java.sun.com/javaee/javaserverfaces/*
18. S. Pietschmann, M. Voigt, A. Rümpel, K. Meissner. CRUISe: Composition of Rich User Interface Services. *ICWE'09*, pp. 473-476.
19. M. Feldmann, T. Nestler, U. Jugel, K. Muthmann, G. Hübsch, A. Schill. Overview of an end user enabled model-driven development approach for interactive applications based on annotated services. *WEWST'09*, pp. 19-28.