Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Distributed orchestration of user interfaces

Florian Daniel [a,*], Stefano Soi [a], Stefano Tranquillini [a], Fabio Casati [a], Chang Heng [b], Li Yan [b]

[a] *Department of Information Engineering and Computer Science, University of Trento Via Sommarive 5, 38123 Povo (TN), Italy*
[b] *Huawei Technologies Shenzhen, PR China*

ARTICLE INFO

Keywords:
UI orchestration
Distributed UIs
UI orchestration patterns
BPEL4UI
Mashups
UI components
MarcoFlow

ABSTRACT

Workflow management systems focus on the coordination of people and work items, service composition approaches on the coordination of service invocations, and, recently, web mashups have started focusing on the integration and coordination of pieces of user interfaces (UIs), e.g., a Google map, inside simple web pages. While these three approaches have evolved in a rather isolated fashion – although they can be seen as evolution of the componentization and coordination idea from people to services to UIs – in this paper we describe a component-based development paradigm that conciliates the core strengths of these three approaches inside a single model and language. We call this new paradigm *distributed UI orchestration*, so as to reflect the mashup-like and process-based nature of our target applications. In order to aid developers in implementing UI orchestrations, we equip the described model and language with suitable design, deployment, and runtime instruments, covering the whole life cycle of distributed UI orchestrations.

1. Introduction

Workflow management systems support office automation processes, including the automatic generation of form-based user interfaces (UIs) for executing human tasks in a process. *Service orchestrations* and related languages focus instead on integration at the application level. As such, this technology excels in the reuse of components and services but does not facilitate the development of UI front-ends for supporting human tasks and complex user interaction needs, which is one of the most time consuming tasks in software development [1].

Only recently, *web mashups* [2] have turned lessons learned from data and application integration into lightweight, simple composition approaches featuring a significant innovation: integration at the UI level. Besides web services or data feeds, mashups reuse pieces of UI (e.g., content extracted from web pages or JavaScript UI widgets) and integrate them into a new web page. Mashups, therefore, manifest the need for reuse in UI development and suitable UI component technologies. Interestingly, however, unlike what happened for services, this need has not yet resulted in accepted component-based development models and practices.

This paper tackles the development of applications that require service composition/process automation logic but that also include human tasks, where humans interact with the system via possibly complex and sophisticated UIs that are tailored to help perform the specific job they want to carry out. In other words, this work targets the development of *mashup-like applications that require process support*, including applications that require distributed mashups coordinated in real time, and provides design and tool support for professional developers, yielding an original composition paradigm based on web-based UI components and web services.

This class of applications manifests a common need that today is typically fulfilled by developing UIs in ad hoc ways and using and manually configuring a process

* Corresponding author. Tel.: +39 0461 283780; fax: +39 0461 282093.
E-mail addresses: daniel@disi.unitn.it (F. Daniel), soi@disi.unitn.it (S. Soi), tranquillini@disi.unitn.it (S. Tranquillini), casati@disi.unitn.it (F. Casati), changheng@huawei.com (C. Heng), liyanmr@huawei.com (L. Yan).

0306-4379/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2011.08.001

engine in the back-end for process automation. As an example, consider the *scenario* in Fig. 1: the figure shows a home assistance application for the Province of Trento whose development we want to aid in one of our projects. A patient can ask for the visit of a home assistant (e.g., a paramedic) by calling (via phone) an *operator* of the assistance service. Upon request, the operator inputs the respective details and inspects the patient's data and personal health history in order to provide the assistant with the necessary instructions (steps 1–5). There is always one assistant on duty. The *home assistant* views the description, visits the patient, and files a report about the provided service (steps 6 and 7). The report is processed by the back-end system and archived (steps 8 and 9). If no further exams are needed, the process ends (steps 10 and 11). If exams are instead needed, the operator books the exam in the local hospital asking confirmation to the patient via phone (steps 12 and 13). Upon confirmation of the exam booking, the system also archives the booking, which terminates the responsibility of the home assistance service (steps 14 and 15).

The application in the scenario includes, besides the process logic, two mashup-like, web-based control consoles for the operator and the assistant that are themselves part of the orchestration, need to interact with the process, and are affected by its progress. In addition, the UIs are themselves component-based and created by reusing and combining existing UI components that are instantiated in the users' web browsers (both web pages in Fig. 1 are composed of four components). The two applications, once instantiated, allow the operator and assistant to manage an individual request for assistance; each new request requires starting a new instance of the application.

In summary, the scenario requires the coordination of the individual actors in the process and the development of the necessary distributed user interface and service orchestration logic. Doing so requires addressing a set of *challenges* (each leading to a specific contribution):

1. Understanding how to *componentize UIs* and *compose* them into web applications.
2. Defining a logic that is able to *orchestrate both UIs and web services*.
3. Providing a *language and tool* for implementing distributed UI compositions and
4. Developing a *runtime environment* that is able to execute distributed UI and service compositions.

This article is an *extended version* of our paper [3] presented at the BPM 2010 conference, in which we approached these challenges in their core aspects. Here, we advance that work in several ways: we provide a complete description of the nature of UI components and of the development and configuration of layout templates, turning the paper into a self-contained piece of work. We conceptualize the types of orchestrations that can be developed with the described development paradigm and discuss their impact on the runtime platform; this represents a major new contribution. We describe our

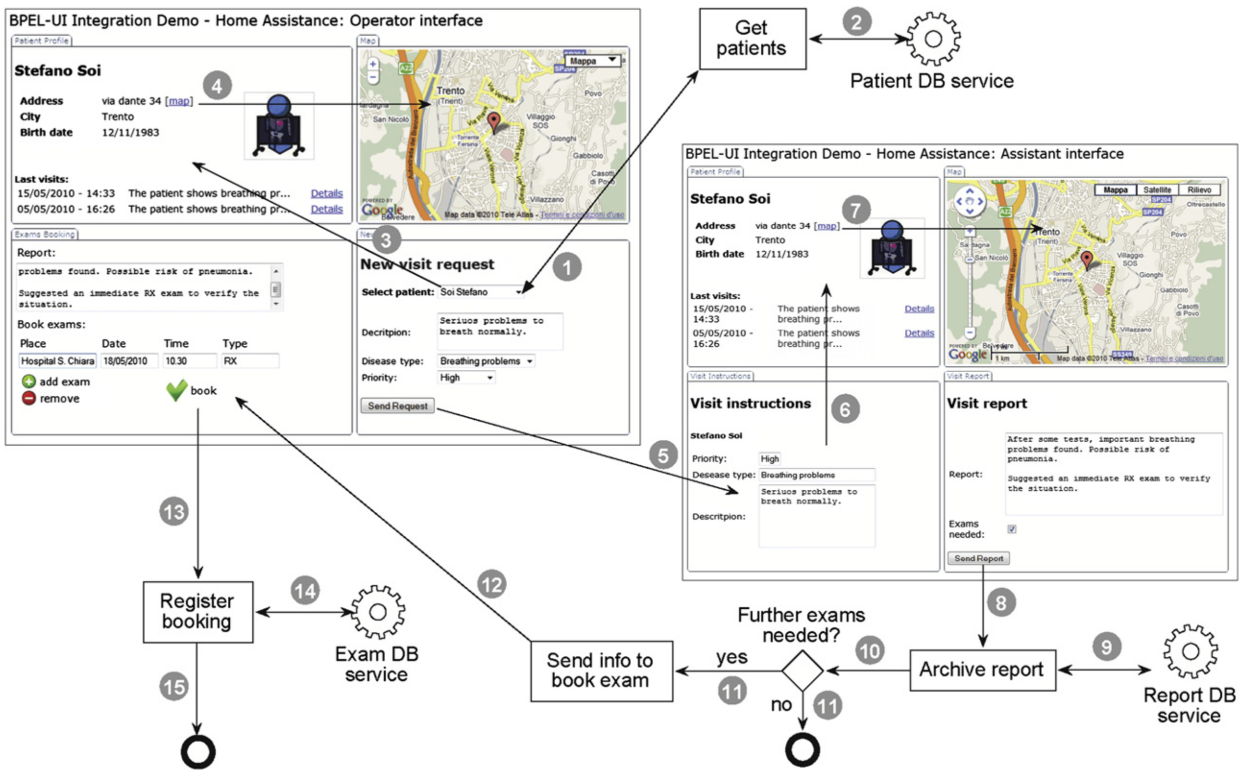

BPEL-UI Integration Demo - Home Assistance: Operator interface
Stefano Soi
Address via dante 34 [map]
City Trento
Birth date 12/11/1983
Last visits:
15/05/2010 - 14:33 The patient shows breathing pr... Details
05/05/2010 - 16:26 The patient shows breathing pr... Details
Report:
problems found. Possible risk of pneumonia.
Suggested an immediate RX exam to verify the situation.
Book exams:
Place Date Time Type
Hospital S. Chiara 18/05/2010 10.30 RX
add exam
remove
book
New visit request
Select patient: Soi Stefano
Decritpion: Seriuos problems to breath normally.
Disease type: Breathing problems
Priority: High
Send Request
Get patients
Patient DB service
BPEL-UI Integration Demo - Home Assistance: Assistant interface
Visit instructions
Priority: High
Desease type: Breathing problems
Descritpion: Seriuos problems to breath normally.
Visit report
Report: After some tests, important breathing problems found. Possible risk of pneumonia. Suggested an immediate RX exam to verify the situation.
Exams needed:
Send Report
Register booking
Exam DB service
Send info to book exam
Further exams needed?
yes
no
Archive report
Report DB service
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15

Fig. 1. A home assistance application integrating both web services and UI components into a process-like orchestration logic.

BPEL4UI editor and the web-based management console, and, finally, we summarize the lessons that we learned during the development and use of the described system.

In Section 2 we introduce the state of the art of the related composition approaches and technologies. In Section 3, we derive requirements from the above scenario and outline the approach we follow in this paper, including the architecture of our *MarcoFlow* platform that will serve as a guide throughout the rest of the paper. In Section 4, we then introduce the concept of HTML/JavaScript UI component and show how defining a new type of binding allows us to leverage the standard WSDL [4] language to abstractly describe them. We then build on existing composition languages (in particular WS-BPEL [5]) to introduce the notions of UI components, pages, and actors into service compositions (Section 5) and explain how such extension can be used to model UI orchestrations (Section 6). In Section 7 we discuss the different types of UI orchestrations that can be implemented. In Section 8, we show how we extended the Eclipse BPEL editor to support design, and we describe how to run UI orchestrations. Finally, in Section 9 we report on the lessons we learned with MarcoFlow and conclude the paper in Section 10.

2. State of the art in orchestrating services, people and UIs

Workflow or business process management systems are the traditional solution to coordinate people; web services have been integrated over the last decade, while support for UI development is still rather weak. For instance, the Oracle BPEL Proccess Manager (http://www.oracle.com/technetwork/middleware/bpel) uses Workflow Services to handle the work-lists of each user and to allow them to perform their tasks. The tool provides two solutions for creating user interfaces: automatic generation, where the tool generates the forms, and custom generation, which enables the modeler to select the template and the parameters to display. Both solutions produce a JSP-based form. Bonita Studio (http://www.bonitasoft.com) has an extension of the tool to create forms. The software allows the developer to use existing form templates; alternatively, forms can be created using a WYSIWYG interface. Forms can be customized by hand and exported as portlets. Similarly, also the tool based on the popular workflow language YAWL [6] and its extension (YAWL4Film [7]) do not go beyond custom or automatically generated web forms (based on the Java Server Faces technology). WebRatio BPM [8] allows the developer to generate WebML [9] web application templates starting from BPMN process models. The templates can then be refined by the developer to equip each page (for task execution) with the necessary data and application functionality, which enables the tool to automatically generate the necessary application code.

All these solutions provide good means to render input and output parameters of tasks as HTML forms, which can either be based on pre-defined form templates or custom forms implemented by the developer. None of the approaches, however, supports the reuse of third-party UIs (e.g., a Google map) as first-class application components and, hence, they are not able to orchestrate them. The synchronization of the two pages in our reference scenario, requiring direct UI-to-UI communications, is thus out of the reach of these tools.

In *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 invocation of REST services [10] or at exposing BPEL processes as REST services [11]. IBM's Sharable Code platform [12] 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 filled at runtime with dynamically generated content.

BPEL4People [13] 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 [14] specification, which focuses on the definition of human tasks, including their properties, behavior and operations used to manipulate them. BPEL4People supports people activities in the 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 [15] and VisualWade [16]. 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 [17], OOHDM [18], and UWE [19]. 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 [20] and process-based web applications [21]; reuse is however limited to web services and UIs are generated out of dynamically filled HTML templates.

A first approach to *component-based UI development* is represented by portals and portlets [22], 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., in (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; also the WSRP [23] specification only provides support for accessing remote portlets as web services.

Finally, the *web mashup* [2] community has 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 that interprets annotations inside web pages supporting the personalization of web pages with UI widgets; service composition is outside the scope of Mash Maker.

In the mashArt [24] 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 and complex features like transactions or exception handling are outside its scope. The CRUISe project [25] 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 [26] aims to support 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.

As this analysis shows, existing development approaches for web-based applications lack an integrated support for service orchestration, component-based UI development, *and* coordination of users, three ingredients that instead are necessary to fully implement applications like the one described in our example scenario.

3. Distributed user interface orchestration: definitions, requirements, and architecture

If we analyze the home assistance scenario, we see that the envisioned application (as a whole) is highly distributed over the Web: the UIs for the actors participating in the application are composed of UI components, which can be components developed in-house (like the `Patient Profile` component) or sourced from the Web (like the `Map` component); service orchestrations are based on web services. The UI exposes the state of the application and allows users to interact with the application and to enact service calls. The two applications for the operator and the assistant are instantiated in different web browsers, contributing to the distribution of the overall UI and raising the need for synchronization.

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*. We call an application that is able to manage these two layers in an integrated fashion a *distributed UI orchestration* [3].

3.1. Requirements and approach

Supporting the development of distributed UI orchestrations is a complex and challenging task. Especially the aim of providing a development approach that is able to cover all development aspects in an *integrated* fashion poses requirements to the whole life cycle of UI orchestrations, in particular, in terms of design, deployment, and execution support.

Indeed, supporting the *design* of distributed UI orchestrations requires:

- Defining a new type of component, the *UI component*, which is able to modularize pieces of UI and to abstract their external interfaces. For the description of UI components, we slightly extend WSDL [4], obtaining what we call *WSDL4UI*, a language that is able to deal with the novel technological aspects that characterize UI components by reusing the standard syntax of WSDL.
- Bringing together the needs of *UI synchronization and service orchestration* in one single language. UIs are typically event-based (e.g., user clicks or key strokes), while service invocations are coordinated via control flows. In this paper, we show how to extend the standard BPEL [5] language in order to support UIs. We call this extended language *BPEL4UI*.
- Implementing a suitable, *graphical design environment* that allows developers to visually compose services and UI components and to define the grouping of UI components into pages. BPEL comes with graphical editors and ready, off-the-shelf runtime engines that we can reuse. For instance, we extend the Eclipse BPEL editor with UI-specific modeling constructs in order to design UI orchestrations and generate BPEL4UI in output.

Supporting the *deployment* of UI orchestrations requires:

- Splitting the BPEL4UI specification into the *two orchestration layers* for intra-page UI synchronization and

distributed UI synchronization and web service orchestration. For the former we use a lightweight UI composition logic, which allows specifying how UI components are coordinated in the client browser. For the latter we rely on standard BPEL.
- Providing a set of *auxiliary web services* that are able to mediate communications between the client-side UI composition logic and the BPEL logic. We achieve this layer by automatically generating and deploying a set of web services that manage the UI-to-BPEL and BPEL-to-UI interactions.

Supporting the *execution* of UI orchestrations requires:

- Providing a *client-side runtime framework* for UI synchronization that is able to instantiate UI components inside web pages and to propagate events from one component to other components. Events of a UI component may be propagated to components running in the same web page or in other pages of the application as well as to web services.
- Providing a *communication middleware* layer that is able to run the generated auxiliary web services for UI-to-BPEL and BPEL-to-UI communications. We implement this layer by reusing standard web server technology able to instantiate SOAP and RESTful web services.
- Setting up a *BPEL engine*, in charge of orchestrating web services and distributed UI-to-UI communications, and implementing a *management console* for both developers and participants in UI orchestrations, enabling them to deploy UI orchestrations, to instantiate them, and to participate in them as required.

These requirements and the respective hints to our solution show that the main methodological goals in achieving our UI orchestration approach are (i) relying as much as possible on existing standards (to start from a commonly accepted and known basis), (ii) providing the developer with only few and simple new concepts (to facilitate fast learning), and (iii) implementing a runtime architecture that associates each concern with the right level of abstraction and software tool (to maximize reuse), e.g., UI synchronization is handled in the browser, while service orchestration is delegated to the BPEL engine.

3.2. Architecture

A possible system architecture that meets the above requirements is shown in Fig. 2. It is the architecture of our MarcoFlow platform, which has been developed jointly by Huawei Technologies and the University of Trento. For presentation purposes, we discuss a slightly simplified version and partition its software components into design time, deployment time, and runtime components.

The *design* part comprises a BPEL4UI editor, which comes with a UI partner link configurator, enabling the setup of UI components inside a UI orchestration, and a layout configurator, assisting the developer in placing UI components into pages. Starting from a set of web service WSDLs, UI component WSDL4UIs, and HTML templates the application developer graphically models the UI orchestration, and the editor generates a corresponding BPEL4UI specification in output, which contains in a single file the whole logic of the UI orchestration.

The *deployment* of a UI orchestration requires translating the BPEL4UI specification into executable formats. In fact, as we will see, BPEL4UI is not immediately executable neither by a standard BPEL engine nor by the UI rendering engine (the so-called UI engine in the right hand side of the figure). This task is achieved by the BPEL4UI compiler, which, starting from the BPEL4UI specification, the set of used HTML templates and UI component WSDL4UIs, and the system configuration of the runtime part of the architecture, generates three kinds of outputs:

1. A set of *communication channels* (to be deployed in the so-called UI engine server), which mediate communications between the UI engine client (the client browser) and the BPEL engine. These channels are crucial in that they resolve the technology conflict inherently present in BPEL4UI specifications: a BPEL engine is not able to talk to JavaScript UI components running inside a client browser, and UI components are not able to interact with the SOAP interface of a BPEL engine. For each UI component in a page, the compiler therefore generates (i) an event proxy that is able to forward events from the client browser to the BPEL engine and (ii) an event buffer that is able to accept events from the BPEL engine and store them on behalf of the UI engine client. The compiler also generates suitable WSDL files for proxies and buffers.
2. A *standard BPEL specification* containing the distributed UI synchronization and web service orchestration logic (see Section 6.1). Unlike the BPEL4UI specification, the generated BPEL specification does no longer contain any UI-specific constructs and can therefore be executed by any standards-compliant BPEL engine. This means that all references to UI components in input to the compilation process are rewritten into references to the respective communication channels of the UI components in the UI engine server, also setting the correct, new SOAP endpoints.
3. A set of *UI compositions*[1] (one for each page of the application) consisting of the layout of the page, the list of UI components of the page, the assignment of UI components to place holders, the specification of the intra-page UI synchronization logic (see Section 6.1), and a reference to the client-side runtime framework. Interactions with web services or UI components running in other pages are translated into interactions with local system components (the notification handlers and event forwarders), which manage the necessary interaction with the communication channels via suitable RESTful web service calls.

Finally, the BPEL4UI compiler also manages the deployment of the generated artifacts in the respective

[1] Details about the format and logic of these UI compositions can be found in [24].

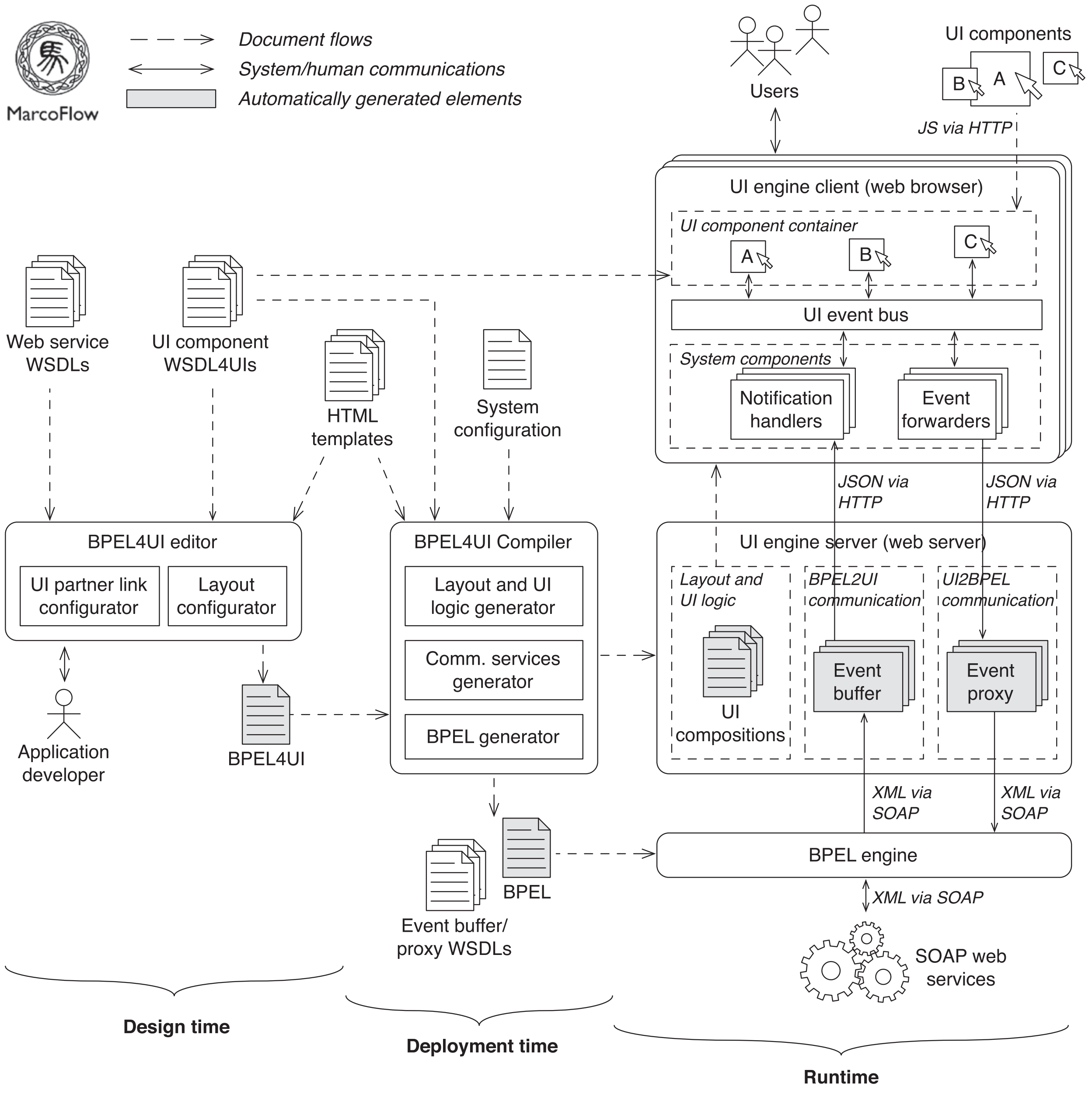


Fig. 2. From design time to runtime: overall system architecture of MarcoFlow.

runtime environments. Specifically, the generated communication channels and the UI compositions are deployed in the UI engine server and the standard BPEL specification is deployed in the BPEL engine.

The *execution* of a UI orchestration requires the setting up and coordination of three independent runtime environments: first, the interaction with the users is managed in the client browser by an event-based JavaScript runtime framework that is able to parse the UI composition stored in the UI engine server, to instantiate UI components in their respective place holders, to configure the notification handlers and event forwarders, and to set up the necessary logic ruling the interaction of the components running inside the client browser. While event forwarders are called each time an event is to be sent from the client to the BPEL engine, the notification handlers are active components that periodically poll the event buffers of their UI components on the UI engine server in order to fetch possible events coming from the BPEL engine.

Second, the UI engine server must run the web services implementing the communication channels. In practice we generate standard Java servlets and SOAP web services, which can easily be deployed in a common web server, such as Apache Tomcat. The use of web server technology is mandatory in that we need to be able to accept notifications from the BPEL engine and the UI engine client, which requires the ability of constantly listening. The event buffer is implemented via a simple relational database (in PostgreSQL, http://www.postgresql.org) that manages multiple UI components and distinguishes between instances of UI orchestrations by means of a session key that is shared among all UI components participating in a same UI orchestration instance.

Third, running the BPEL process requires a BPEL engine. Our choice to rely on standard BPEL allows us to reuse a common engine without the need for any UI-specific extensions. In our case, we use Apache ODE (http://ode.apache.org), which is characterized by a simple deployment procedure for BPEL processes.

We discuss each of the ingredients in the following.

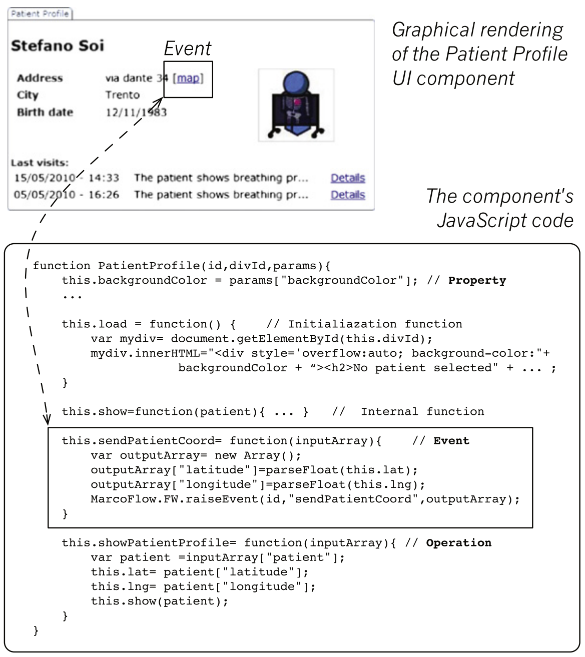
4. The building blocks: web services and UI components

Orchestrating remote application logic and pieces of UI requires, first of all, understanding the exact nature of the components to be integrated, i.e., web services and UI components.

For the *integration of application logic*, we rely on standard web service technologies, such as *WSDL-SOAP web services*, i.e., remote web services whose external interface is described in WSDL, which supports interoperability via four message-based types of operations: request-response, notification, one-way, and solicit-response. Most of today's web services of this kind are stateless, meaning that the order of invocation of their operations does not influence the success of the interaction, while there are also stateful services whose interaction requires following a so-called business protocol that describes the interaction patterns supported by the service.

For the *integration of UI*, we rely instead on *JavaScript/HTML UI components*, which are simple, stand-alone web applications that can be instantiated and run inside any common web browser [24]. Fig. 3 illustrates an example of UI component (the `Patient Profile` UI component of our reference scenario), along with an excerpt of its JavaScript code. The figure shows that, unlike web services, UI components are characterized by:

- A *user interface*: UI components can be instantiated inside a web browser and can be accessed and navigated by a user via standard HTML. The UI allows the user to interactively inspect and alter the content of the component, just like in regular web applications. UI components are therefore stateful, and the component's navigation features replace the business protocol needed for services.
- *Events*: Interacting with the UI generates system events (e.g., mouse clicks) in the browser used to manage the update of contents. Some events may be exposed as component events, in order to communicate state changes. For instance, a click on the "map" link in Fig. 3 launches a `sendPatientCoord` event.
- *Operations*: Operations enact state changes from the outside. Typically, we can map the event of one component to the operation of another component in order to synchronize the components' state (so that they show related information).


Patient Profile
Stefano Soi
Event
Address via dante 34 [map]
City Trento
Birth date 12/11/1983
Last visits:
15/05/2010 - 14:33 The patient shows breathing pr... Details
05/05/2010 - 16:26 The patient shows breathing pr... Details
Graphical rendering of the Patient Profile UI component
The component's JavaScript code

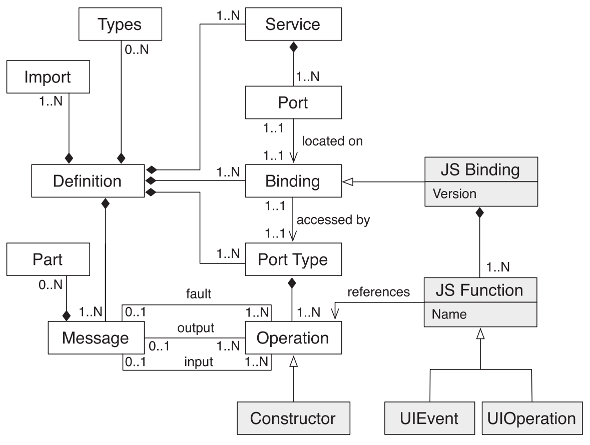
```
function PatientProfile(id,divId,params){
    this.backgroundColor = params["backgroundColor"]; // Property
    ...

    this.load = function() {    // Initialiazation function
        var mydiv= document.getElementById(this.divId);
        mydiv.innerHTML="<div style='overflow:auto; background-color:"+
                  backgroundColor + "><h2>No patient selected" + ... ;
    }

    this.show=function(patient){ ... }   //  Internal function

    this.sendPatientCoord= function(inputArray){    // Event
        var outputArray= new Array();
        outputArray["latitude"]=parseFloat(this.lat);
        outputArray["longitude"]=parseFloat(this.lng);
        MarcoFlow.FW.raiseEvent(id,"sendPatientCoord",outputArray);
    }

    this.showPatientProfile= function(inputArray){ // Operation
        var patient =inputArray["patient"];
        this.lat= patient["latitude"];
        this.lng= patient["longitude"];
        this.show(patient);
    }
}
```

Fig. 3. Graphical rendering and internal logic of a UI component.

- *Properties*: The graphical setup of a component may require the setting of constructor parameters, e.g., to align background colors or set other style properties.

In order to make UI components accessible to BPEL, each component must be equipped with a descriptor that describes its events, operations, and properties in terms of WSDL operations. As already anticipated in the previous section, doing so requires extending the standard WSDL description logic, i.e., its meta-model, from web services to UI components. The result of this extension is called *WSDL4UI*. Fig. 4 illustrates its meta-model, from which we can see that the extension toward UI components occurs via two different techniques:

1. First, we introduce a set of *conventions* of how the abstract WSDL constructs can be used to describe UI components. The properties of the UI component are encapsulated by means of a dedicated *constructor* operation that can be used to set properties at instantiation time of the component. Next, all operations specified in the description are either UIOperations, UIEvents, or a constructor. *UIOperations* have only inputs; *UIEvents* have only outputs; the constructor is an operation. Finally, the *port address* of the described service corresponds to the URL at which the actual UI component can be downloaded for instantiation (in form of a JavaScript file).
2. Second, we introduce a new *JavaScript binding* that allows us to associate to each abstractly defined operation a JavaScript function of the UI component. Doing so enables the client-side runtime environment (the UI engine client) to parse the WSDL4UI description of a



WSDL4UI conventions:
(1) All *Operations* are either *UIOperations*, *UIEvents*, or a *Constructor*.
(2) *UIOperations* only have inputs.
(3) *UIEvents* only have outputs.
(4) The *Constructor* is unique and has only inputs.
(5) The service's *port address* points to the JavaScript class of the UI component.

Fig. 4. Simplified WSDL4UI meta-model (inspired by [27] and extended – via the gray boxes – toward UI components).

component, to invoke its constructor, and to correctly access events and operations in JavaScript.

Only WSDL files that conform to these rules are considered correct WSDL4UI descriptors of UI components. Fig. 5, for instance, shows the descriptor of the `Patient Profile` UI component. Its interface is characterized by three WSDL operations: `ShowPatientProfile`, `SendPatientCoord`, and `constructor` (lines 9–17), corresponding, respectively, to a UIOperation, to a UIEvent and to the component's constructor, as stated in the JavaScript binding (lines 20-31). In the binding, there are also specified, through the related `jsFunction` attributes (e.g., line 23), the actual JavaScript functions implementing the operations, which are contained in the file located at the URL defined in the service's port address (line 35).

For the BPEL engine, in order to interact with a component, the BPEL4UI compiler introduced in Section 3.2 generates a respective event buffer and event proxy for the UI engine server and equips them with two standard WSDL descriptors. These descriptors contain the abstract service description as defined in the WSDL4UI file (the event buffer contains all operations of the UI components, the event proxy all events), yet their port addresses point to the newly generated services and their JavaScript binding is turned into a SOAP binding.

5. The UI orchestration meta-model

Starting from web services and UI components, developing a UI orchestration requires modeling two fundamental aspects: (i) the *interaction logic* that rules the passing of data among UI components and web services and (ii) the *graphical layout* of the final application. Supporting these tasks in service orchestration languages (like BPEL) requires extending the expressive power of the languages with UI-specific constructs.

Fig. 6 shows the simplified meta-model of BPEL4UI, addressing these two concerns. Specifically, the figure details all the new modeling constructs necessary to specify UI orchestrations (gray-shaded) and omits details of the standard BPEL language, which are reused as is by BPEL4UI (a detailed meta-model for BPEL can be found, for instance, in [28]). The code snippet in Fig. 7 exemplifies the syntax that we use, in order to express the novel concepts in BPEL4UI.

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. There can be multiple partner links for each partner link type.

Modeling UI-specific aspects requires instead introducing a set of new constructs that are not yet supported by BPEL. The constructs, illustrated in Fig. 6, are:

- *UI type*: The introduction of UI components into service compositions asks for a new kind of partner link type. Although syntactically there is no difference between web services and UI components (the JavaScript binding introduced into WSDL4UI comes into play only at runtime), it is important to distinguish between services and UI components as (i) their semantics and, hence, their usage in the model will be different from that of standard web services, and (ii) the UI orchestration editor must be aware of whether an object manipulated by the developers is a web service or a UI component, in order to support the setting of UI-specific properties.

  As exemplified in Fig. 7, we specify the new partner link type like a standard web service type (lines 7–10). In order to reflect the events and operations of the UI component, we distinguish the two roles. Lines 1–5 define the necessary name spaces and import the WSDL4UI descriptor of the UI component.
- *Page*: The distributed UI of the overall application consists of one or more web pages, which can host instances of UI components. Pages have a *name*, a *description*, a reference to the pages' *layout template*, the name of the *UI engine* they will run on, and an indication of whether they are a *start page* of the application or not (as we will see in Section 7, inside a process model, not all pages allow the correct instantiation of the process).

  The code lines 13–20 in Fig. 7 show the definition of a page called "operator", along with its layout template and the name of the UI engine on which the page will be deployed; the page is a start page for the process.
- *Place holder*: Each page comes with a set of place holders, which are empty areas inside the layout template that can

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions name="PatientProfile" targetNamespace="http://www.unitn.it/
JS/Patient" ... >
  <!-- types definition -->
  ...
  <!-- massages definition -->
  ...
  <wsdl:portType name="PatientPortType">
      <wsdl:operation name="constructor">
           <wsdl:input message="tns:constructorMessage"/>
       </wsdl:operation>
       <wsdl:operation name="ShowPatientProfile">
           <wsdl:input message="tns:ShowPatientProfileMessage"></wsdl:input>
           </wsdl:operation>
       <wsdl:operation name="SendPatientCoord">
           <wsdl:output message="tns:SendPatientCoordMessage"></wsdl:output>
       </wsdl:operation>
   </wsdl:portType>

   <wsdl:binding name="PatientJS" type="tns:PatientPortType">
       <js:binding version="1.0" />
       <wsdl:operation name="constructor">
           <js:operation jsFunction="load" />
       </wsdl:operation>
       <wsdl:operation name="ShowPatientProfile">
           <js:operation jsFunction="showPatientProfile" />
       </wsdl:operation>
       <wsdl:operation name="SendPatientCoord">
           <js:event jsFunction="sendPatientCoord" />
       </wsdl:operation>
   </wsdl:binding>

   <wsdl:service name="PatientProfile">
       <wsdl:port name="PatientJS" binding="tns:PatientJS">
           <soap:address location="http://www.unitn.it/JS/Patient.js" />
       </wsdl:port>
   </wsdl:service>
</wsdl:definitions>
```

Fig. 5. Example of WSDL/UI description of a UI component.

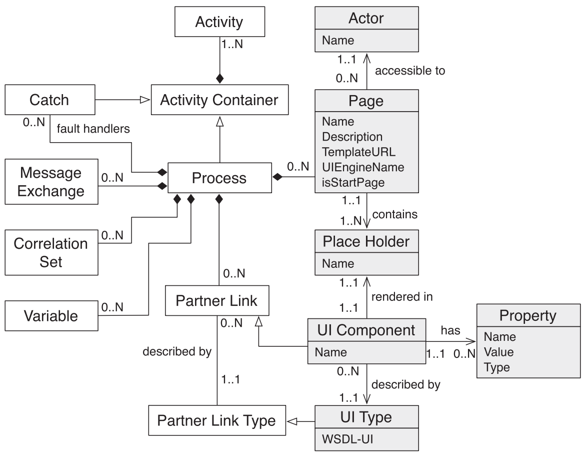


Fig. 6. Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs [28]; gray classes correspond to constructs for UI and user management.

be used for the graphical rendering of UI components. Place holders are identified by a unique *name*, which can be used to associate UI components.

Place holders are associated with page definitions and specified as sub-elements, as shown in lines 16–19 in Fig. 7.

- *UI component*: UI types can be instantiated as UI components. For instance, there may be one UI type but two different instances of the type running in two different web pages. Declaring a UI component in a BPEL4UI model leads to the creation of an instance of the UI component in one of the pages of the application. Each component has a unique *name*.

  We specify UI component partner links by extending the standard partner link definition of BPEL with three new attributes, i.e., *isUiComponent*, *pageName*, and *placeHolderName*. Lines 25–32 in Fig. 7 show how to declare the `Patient Profile` component of our example scenario.
- *Property*: As we have seen in the previous section, UI components may have a constructor that allows one to set configuration properties. Therefore, each UI component may have a set of associated properties than can be parsed at instantiation time of the component. We use simple *name-value* pairs to store constructor parameters.

  Properties extend the definition of UI component link types by adding property sub-elements to the partner link definition, one for each constructor parameter, as shown in lines 30–31 in Fig. 7.
- *Actor*: In order to coordinate the people in a process, pages of the application can be associated with individual actors, i.e., humans, which are then allowed to access the page and to interact with the UI orchestration via the UI components rendered in the page. As for now, we simply associate static actors to pages (using their *names*); yet, actors can easily be assigned

```
 1 <bpel:process name="HomeAssistance" targetNamespace="http://www.unitn.it/
 2 example/HomeAssistance" xmlns:wsdl6="http://www.unitnt.it/JS/Patient" ...>
 3  <bpel:import namespace="http://www.unitnt.it/JS/Patient"
 4                location="Patient.wsdl" importType="http://
 5                schemas.xmlsoap.org/wsdl/" />
 6  ...
 7  <bpel:partnerLinkType name="PatientPL">
 8      <bpel:role name="receive" portType="wsdl6:PatientPortTypeReceive"/>
 9      <bpel:role name="invoke" portType="wsdl6:PatientPortTypeInvoke"/>
10  </bpel:partnerLinkType>
11  ...
12  <bpel4ui:pages>
13      <bpel4ui:page name="operator" templateURL="operator.html"
14                 uiEngineName="HAEngine" actorName="SteS"
15                 description="the operator page" isStartPage="true" >
16          <bpel4ui:placeHolder name="marcoflow-top-left" />
17          <bpel4ui:placeHolder name="marcoflow-top-right" />
18          <bpel4ui:placeHolder name="marcoflow-bottom-left" />
19          <bpel4ui:placeHolder name="marcoflow-bottom-right" />
20      </bpel4ui:page>
21      ...
22  </bpel4ui:pages>
23
24  <bpel:partnerLinks>
25      <bpel:partnerLink name="PatientProfileUI_operator"
26                        partnerLinkType="tns:PatientPL"
27                        myRole="receive" partnerRole="invoke"
28                        isUiComponent="yes" pageName="operator"
29                        placeholderName="marcoflow-top-left">
30          <bpel4ui:property name="backgroundColor" type="xsd:string"
31                         value="white" />
32      </bpel:partnerLink>
33      ...
34  </bpel:partnerLinks>
35
36  <!-- orchestration logic definition -->
37  ...
38 </bpel:process>
```

Fig. 7. Excerpt of the BPEL4UI home assistance process (new constructs in bold).

also dynamically at deployment time or at runtime by associating roles instead of actors and using a suitable user management system.

Actors are simply added to page definitions by means of the *actorName* attribute, as highlighted in line 14 in Fig. 7.

The addition of these new concepts to BPEL turns the service orchestration language into a language that, in addition to service invocation logic, is also able to specify the organization of an application's UI and its distribution over multiple servers and actors. Our goal in doing so was to keep the number of new concepts as small as possible, while providing a fully operational specification language for UI orchestrations.

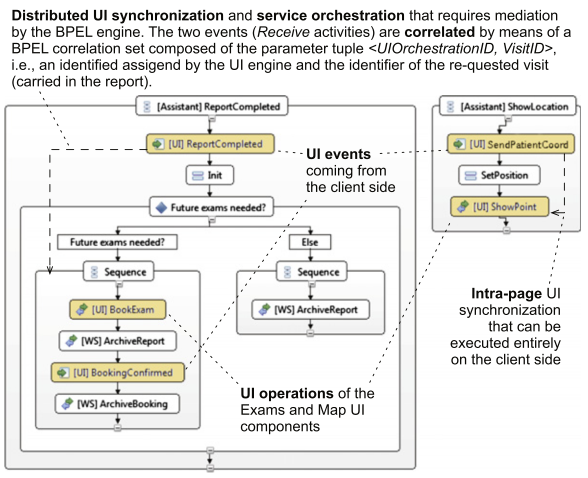
6. Modeling distributed UI orchestrations

The code example in Fig. 7 shows that the UI-specific modeling constructs have a very limited impact on the syntax of BPEL and are mostly concerned with the abstract specification of the layout and the declaration of UI partner links. The actual composition logic, instead, relies exclusively on standard BPEL constructs. Yet, since UI components are different from web services (e.g., it is important to know in which page they are running), modeling UI orchestrations requires a profound understanding of the necessary modeling constructs and their semantics. In particular, it is important to understand the effect that individual modeling patterns have on the execution of the final application, i.e., the semantics of the patterns, and which other modeling tasks (data transformations, message correlations, and layout design) are necessary to fully specify a working UI orchestration.

6.1. *Core UI orchestration design patterns*

The first step toward this understanding is mastering the core design patterns that characterize UI orchestrations. As hinted at in Section 3 and illustrated in Fig. 8, we distinguish three main design patterns:

- *Intra-page UI synchronization*: The small model block (a BPEL *sequence* construct) in the right part of Fig. 8 shows the internals of step 7 in Fig. 1. When the assistant clicks on the "map" link, the patient's address is shown on the Google map. In BPEL terms, we receive a message from the `Patient Profile` UI component (the event) and forward it to the operation of the `Map` component, both running inside the web page of the assistant. The pattern, hence, implements a so-called *intra-page UI synchronization*, i.e., a synchronization of UI components that run inside a same page. From a


Distributed UI synchronization and **service orchestration** that requires mediation by the BPEL engine. The two events (*Receive* activities) are **correlated** by means of a BPEL correlation set composed of the parameter tuple *<UIOrchestrationID, VisitID>*, i.e., an identified assigend by the UI engine and the identifier of the re-quested visit (carried in the report).

UI events coming from the client side

Intra-page UI synchronization that can be executed entirely on the client side

UI operations of the Exams and Map UI components

Fig. 8. Part of the BPEL4UI model of the home assistance process as modeled in the extended Eclipse BPEL editor (the dashed and dotted lines/arrows have been overlaid as a means to explain the model).

runtime point of view, this kind of UI synchronization can be performed entirely on the client side without requiring support from the BPEL engine.

- *Distributed UI synchronization*: The bigger model block (again a BPEL *sequence* construct) in the left part of the figure, instead, contains a *distributed UI synchronization* that cannot be executed on the client side only, as the two UI components involved in the communication (`Visit Report` and `Exams Booking`) run in different web pages. The event generated upon submission of a new report is processed by the BPEL engine, which then decides whether an additional exam needs to be booked by the operator or not. As such, the BPEL engine manages two independent concerns, i.e., the forwarding of the event from one UI component to another and the evaluation of the condition, of which only the former is necessary to implement a distributed UI synchronization pattern. The execution of a distributed UI synchronization pattern always requires the cooperation of both the BPEL engine and the client-side runtime environment.
- *Service orchestration*: The distributed UI synchronization also involves the orchestration of the `Report DB` and `Exam DB` web services, as well as some BPEL flow control constructs. In fact, the modeled logic checks whether the report expresses the need for further exams or not. In either case, the further processing of the report involves the invocation of either one or both the web services, in order to correctly terminate the handling of a visit request. The pure invocation of web services represents a service invocation pattern, whose execution can be entirely managed by the BPEL engine without requiring support from the client-side runtime environment.

The BPEL4UI excerpt in Fig. 8 shows that, when modeling a UI orchestration, it is important to keep in mind who communicates with whom and which UI component will be rendered where. Depending on these two considerations, the modeled composition logic will either be executed on the client side, in the BPEL engine, or in both layers. For instance, it suffices to associate the `Map` component with a different page, in order to turn the intra-page UI synchronization in the right hand side of Fig. 8 into a distributed UI synchronization and, hence, to require support from the BPEL engine.

6.2. Data transformations

When composing services or UI components, it is not enough to model the communication flow only. An important and time-consuming aspect is that of transforming the data passed from one component to another. With BPEL4UI we support all data transformation options provided by BPEL by means of its `Assign` construct. This allows us to leverage on technologies, such as XPath, XQuery, XSLT, or Java, for the implementation of also very complex data transformations.

Yet, it is important to keep in mind that the *type* of data transformation may affect the logic of the UI orchestration: for instance, if the `SetPosition` activity in the top-right corner of Fig. 8 does not transform data at all or only performs simple parameter mappings (with the BPEL `Copy` construct), we fully support the execution of the intra-page UI synchronization in the client browser. If instead a more complex transformation is needed, we rely on the BPEL engine to perform it.

The reason for this choice is that UI synchronization typically requires the exchange of only simple data (e.g., parameter-value pairs), which do not require complex transformation capabilities like the ones we need when interacting with web services. Supporting only simple parameter–parameter mappings on the client side allows us to keep the client-side runtime framework as lightweight as possible, without however giving up any of BPEL's data transformation capabilities.

6.3. Message correlation

Independently of the format of data, UI orchestrations may require a careful design of the messages used in the orchestration and of how these must be *correlated*, in order to enable the runtime environment to dispatch each message to its correct UI orchestration instance. In fact, just like in conventional workflow or service orchestration engines, there may be multiple instances of UI orchestrations running concurrently in a same BPEL/UI engine. Message correlation is required in all those cases where the orchestration involves *multiple entry points* into the orchestration logic (e.g., callbacks from external web services or a condition that requires input from two different events).

If we look at our modeling example in Fig. 8, we see that the intra-page UI synchronization in the top-right corner does not involve multiple entry points. It is therefore not necessary to implement any correlation logic in BPEL4UI, in order to propagate the `SendPatientCoord` event from the `Patient Profile` UI component to the `ShowPoint` operation of the `Map` UI component. Since both UI components involved in this synchronization run inside the same web page and, therefore, there is no ambiguity regarding which instance of the Map UI component is the target of the SendPatientCoord event. In Section 7, we will see that this is not always the case.

The distributed UI synchronization, instead, involves two UI events from two different actors and, hence, different pages: ReportCompleted and BookingConfirmed. In this case, it is necessary to configure a so-called *correlation set* (in BPEL terminology) that allows the BPEL engine to understand when two instances of those events belong to a same process instance. In the example in Fig. 8, we use UIOrchestrationID (provided by the UI engine) and VisitID (part of the report) as correlation set.

6.4. Graphical layout

Finally, the complete definition of a UI orchestration also requires the design of suitable *HTML templates* and the assignment of UI components to their place holders inside the pages. As our goal is the development of an enabling middleware layer for UI orchestrations, for the layout templates we rely on standard web design instruments and technologies (e.g., Adobe Dreamweaver). The only requirement the templates must satisfy is that they provide place holders in the form of HTML DIV elements that can be indexed via standard HTML identifiers following a predefined naming convention: <div id="marcoflow-..."></div>.

Fig. 9, for instance, depicts the empty HTML template of the assistant's web page, whose filled version we have already seen in Fig. 1. The template is a simple HTML page with a page title and the four uniquely identified placeholders to be filled with UI components at runtime. Differently from dynamic HTML and most of the approaches discussed in Section 2, in which the template typically also contains the formatting logic for the data to be rendered inside the place holders, in our case the template only identifies the location of the UI components; the rendering of content is then managed autonomously by the UI components.

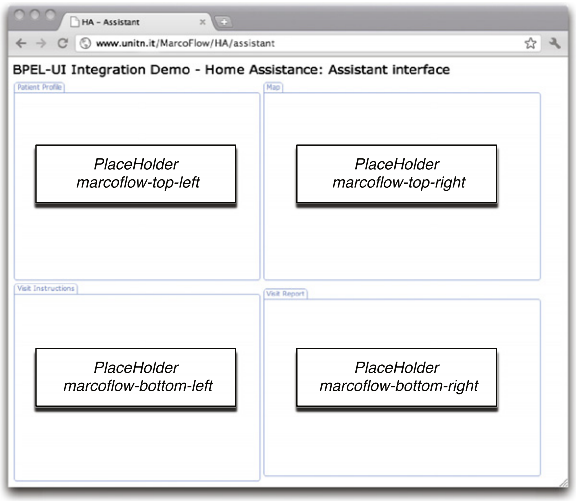

HA – Assistant
www.unitn.it/MarcoFlow/HA/assistant
BPEL-UI Integration Demo - Home Assistance: Assistant interface
Patient Profile
Map
Visit Instructions
Visit Report
PlaceHolder marcoflow-top-left
PlaceHolder marcoflow-top-right
PlaceHolder marcoflow-bottom-left
PlaceHolder marcoflow-bottom-right

Fig. 9. The HTML template of the assistant's web page highlighting the empty place holders for UI components.

Once all HTML templates for all pages in the UI orchestration are defined, the definition of the pages and the association of UI partner links with place holders therein proceeds as exemplified in Section 6.

7. Types of UI orchestrations

So far we have seen how BPEL4UI supports the development of distributed UI orchestrations. Yet, developing correct UI orchestrations is still a non-trivial task, in that the distribution of UI synchronizations and service orchestrations over two different runtime engines (the UI engine and the BPEL engine) complicates the instantiation logic of distributed UI orchestrations, an aspect that developers should understand thoroughly. As illustrated in Fig. 10, we identify four main types of UI orchestrations that can be implemented by means of the core patterns described in Section 6.1, i.e., *pure UI synchronizations*, *pure service orchestrations*, *UI-driven UI orchestrations*, and *process-driven UI orchestrations*. The developer needs to master these configurations if he does not want to encounter unexpected behaviors or errors at runtime. We discuss each of these configurations next.

7.1. Pure UI synchronizations

From a UI point of view, the basic type of UI orchestration is represented by applications that involve *UI components only* and, hence, exclusively focus on the synchronization of UIs via events. Typical examples of this type of UI orchestration are UI-based mashups, portlets/portals, applications that integrate widgets/gadgets, or similar component-based UI applications.

Fig. 10(a) illustrates a simple example: there are two concurrent pages, possibly associated with two different users and with a total of three UI components, one in Page 1 and two in Page 2. By interacting with the UI component A, the user can generate an event that synchronizes component B in the other page; likewise, another user can interact with B and synchronize both A and C, while C allows the user to synchronize again B. The three UI components are instantiated in their web pages and run until the users close their web browsers or navigate to another web page. As such, UI components are stateful: their UI constantly reflects the interaction state of the users with the component (e.g., in terms of selections or navigation actions performed). During their lifetime, each UI component may generate multiple events as output and accept multiple events as input. That is, while in one instance of the UI orchestration in Fig. 10(a) each UI component is instantiated only once, there may be multiple instances of synchronization events (the dashed arrows).

Supporting the *execution* of this type of UI orchestration requires the presence of both a client-side runtime environment and a server-side environment. Specifically, the intra-page UI synchronization of B and C can be handled in the client, since both UI components run inside the same web page, i.e., web browser. The synchronization of A and B, instead, requires help from the server side, in that they implement a distributed UI synchronization. Therefore, the

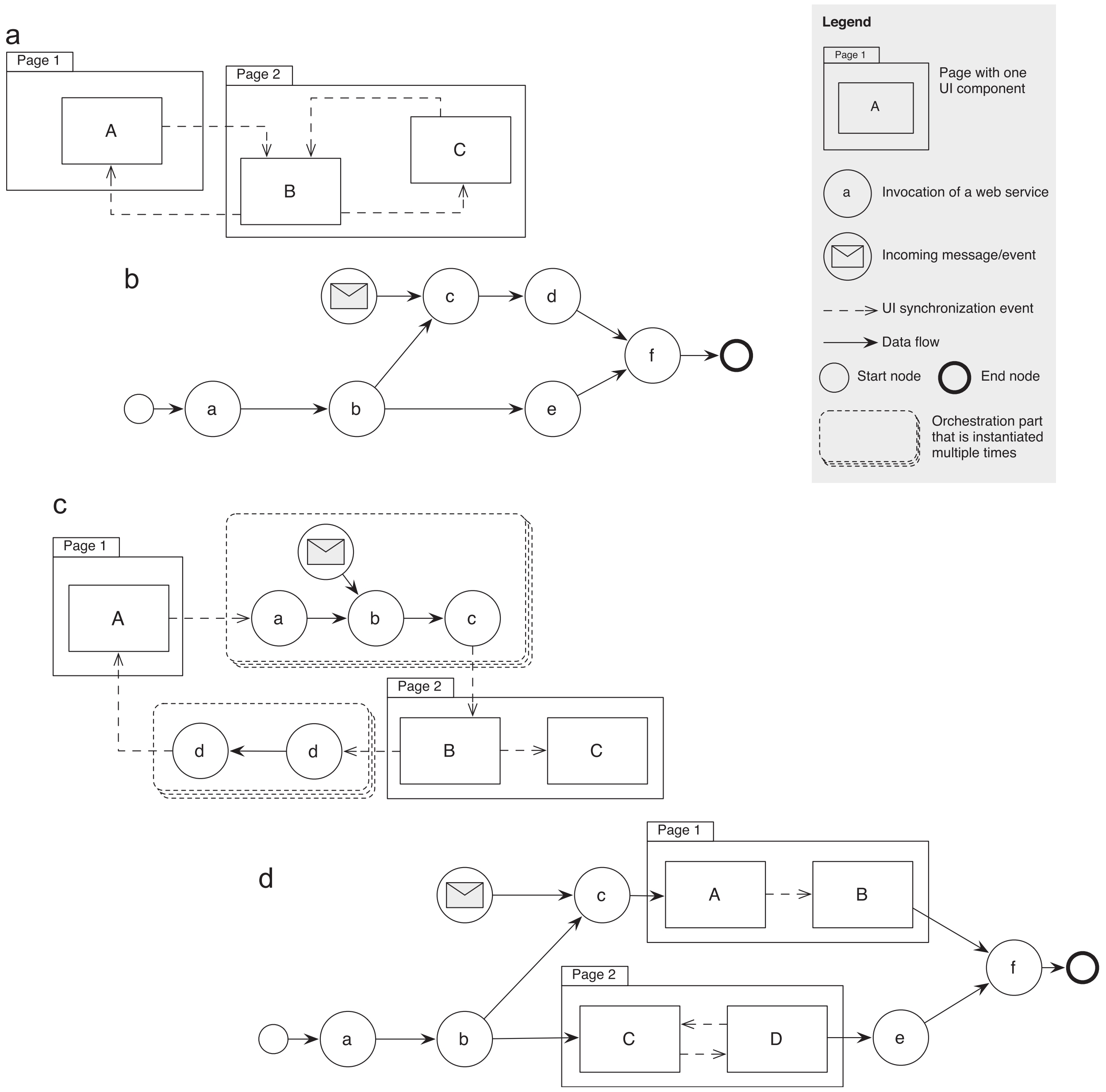


Fig. 10. The four types of (UI) orchestration supported by BPEL4UI and the MarcoFlow system. (a) *Pure UI synchronization* of multiple UI components. (b) *Pure service orchestration* of multiple web service invocations. (c) *UI-driven UI orchestration* with UI components triggering the execution of service orchestration instances. (d) *Process-driven UI orchestration* with the process instance enabling/disabling the access to pages.

event proxy on the server side (cf. Fig. 2) is needed, in order to forward communications among the two web pages.

Sending an event through the event proxy raises the need for *correlation*, in that there may be multiple instances of a same UI orchestration running concurrently and, therefore, it is necessary to identify which event belongs to which instance. The solution we adopt is to add to each generated UI event a so-called *UIOrchestrationID*, which uniquely identifies the UI orchestration instance. The identifier is generated by the UI engine at application startup and shared with all the users participating in the orchestration. This feature is automated in our runtime framework and does not require any specific modeling at design time.

7.2. Pure service orchestrations

From a web service point of view, the basic type of UI orchestration is the one that completely comes *without UI*, i.e., a common web service orchestration. Although this configuration represents a “degenerated” UI orchestration (given that there is no UI), it is fully supported by BPEL4UI and deserves an explanation in that it represents the building block for the next UI orchestration types. Typical examples are order processing logics or payment processes.

Fig. 10(b) provides an example: there are six web service invocations (specifically, synchronous request-response invocations) and one incoming event arranged in a typical service orchestration. For presentation purpose, we adopt a *data flow* logic to model the orchestration, as for the discussion in this section it is not important to explicitly distinguish between control and data flow. The important aspect of the model is that, upon instantiation of the service orchestration, each element in the model is instantiated exactly once—including the data flow connectors (differently from what happened with the UI synchronization events in Fig. 10(a)). The data flow connectors rule both which service invocation can be performed and how data are passed from one invocation to another.

Executing such a service orchestration requires support from an orchestration engine/server, such as a BPEL engine, which is able to instantiate on orchestration model, to invoke the services as prescribed by the model, to transform data formats between service invocations, to accept incoming notifications or events, and to keep the state of the progress in the orchestration instance. The actual services run remotely and are outside the scope of the orchestration environment.

The important aspect of the model in Fig. 10(b) is the incoming event (graphically represented by the letter in the circle), as the event raises the need for *correlation* in the service orchestration. In fact, without the incoming event, the model would consist only of synchronous service invocations, which could be processed easily step by step by the orchestration engine. The engine would simply invoke a service, wait for its response, pass the response to the next service, and so on till the whole orchestration logics ends. In the presence of the incoming event, instead, the engine must be able to correlate each incoming event it receives with the correct target orchestration instance of the event. Doing so requires sharing at least a simple key or identifier (the *correlation set*) among the running orchestration instance and the incoming event. For instance, the name of the person who starts the orchestration instance could be used as correlation identifier, as such could be known to both the engine and the external service sending the event—provided that there is always only one instance per person running in the engine.

7.3. UI-driven UI orchestrations

A "full" UI orchestration, however, is characterized by the joint use of both UI synchronizations and service orchestrations inside a same application. Depending on which of these two ingredients dominates the behavior of the application, we can have either UI-driven orchestrations (where service orchestrations are enacted by the UI) or process-driven orchestrations (where the UIs are enacted by the service orchestration). Here we focus on the former type, in the next section we discuss the latter. For instance, a web mashup that integrates RSS data from a Yahoo! Pipe may invoke the pipe processing logic multiple times while running.

Fig. 10(c) abstracts this type of UI orchestration: there are two pages with respective UI components and two service orchestration flows. While the intra-page UI synchronization of `B` and `C` does not involve any web service, the distributed UI synchronizations of `A` and `B` are based on intermediate service invocations in both directions. Just like we can have multiple UI synchronization events (the dashed arrows) for each instance of UI component, we now also have for each synchronization of `A` and `B` a new instance of the intermediate service orchestration logic (graphically represented by the dashed box around the service orchestrations).

In order to *execute* such a UI-driven UI orchestration, we need to join also the power of the runtime environments of the two previous configurations. Specifically, UI synchronizations involving service invocations can no longer be performed with a simple event proxy on the server side only (like in pure UI orchestrations); instead, the synchronization requires a tight integration of the client-side runtime environment for UIs with the server-side service orchestration engine. Specifically, a UI synchronization event from one page must be able to instantiate and provide input to a service orchestration logic on the server side, which, in turn, must be able to deliver its output in form of a UI synchronization event sent to another page. That is, we need to have a full two-way communication channel between the two runtime environments, a feature that is implemented by the UI components' event proxies and event buffers in the UI engine server.

In terms of *correlation*, all UI synchronization events carry the `UIOrchestrationID`, as already introduced for pure UI orchestrations, while the service orchestration parts may require additional correlation information inside BPEL4UI, depending on their individual topology. For instance, the service orchestration enacted by propagating an event from `B` to `A` only involves synchronous service invocations and does therefore not require any additional correlation information. The other service orchestration in Fig. 10(c), instead, also involves the reception of an external event, which requires the setup of an additional correlation identifier, as already described for Fig. 10(b).

7.4. Process-driven UI orchestrations

Finally, we have a process-driven UI orchestration each time we have an application that brings together UI synchronizations and service orchestrations in which the service orchestration dominates over the UI synchronization. For instance, workflow management or, more in general, business process management applications that integrate both web services and UI components and that orchestrate tasks (work items) to be performed by either users or automated resources, such as our reference scenario, can be considered of this type of UI orchestration.

Fig. 10(d) schematically illustrates the situation: the application starts with a pure service orchestration that enacts a set of services and, only after the successful processing of services a, b, c, and d, allows the users to access their respective web pages. Inside the pages, there are UI components that allow the users to interact with the pages and to perform and conclude their tasks, which causes the UI orchestration to leave again and disable the pages and to proceed with the processing of the remaining part of the service orchestration. That is, in process-driven UI orchestrations pages are invoked like services, but they are targeted at users and, therefore, expose a UI the users can interact with. The overall UI orchestration keeps waiting until the user successfully completes his/her task, which is communicated via an outgoing UI synchronization event.

In terms of required *execution* support, process-driven UI orchestrations are similar to UI-driven UI orchestrations, with the difference that the main service orchestration is instantiated only ones, not multiple times.

Correlation requirements are similar, too. As shown in Fig. 10(d), if there is an incoming event that needs to be injected into a running instance of the UI orchestration, correlation is needed; otherwise, the whole UI orchestration can also be processed without correlation. UI synchronization events are again managed via the orchestration's unique identifier associated by the UI engine.

7.5. Complex UI orchestrations

The four types of UI orchestrations above represent those classes of UI orchestrations that characterize the most important application scenarios we encountered throughout the development of the MarcoFlow system. Yet, UI orchestrations may easily also get more complex. For instance, it is possible to use a process-driven UI orchestration (including again UIs and actors) in place of any of the simple service orchestrations in Fig. 10(c), or it is possible to expand the simple pages in Fig. 10(d) into complete UI-driven UI orchestrations (including new service orchestrations), or we could establish UI synchronizations among the two pages in Fig. 10(d), and similar. While these kinds of UI orchestrations are theoretically possible and supported by BPEL4UI and MarcoFlow, luckily it is hard to find practical examples that indeed require such a level of complexity.

8. Implementing and running UI orchestrations

In order to ease the development, deployment, and execution of UI orchestrations, MarcoFlow comes with two tools that aid the different actors involved: a *graphical BPEL4UI editor* for developers and a *web-based management console* for both developers and users.

The *graphical BPEL4UI editor* for developers has been implemented as an extension of the Eclipse BPEL editor (http://www.eclipse.org/bpel/) and comes 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.

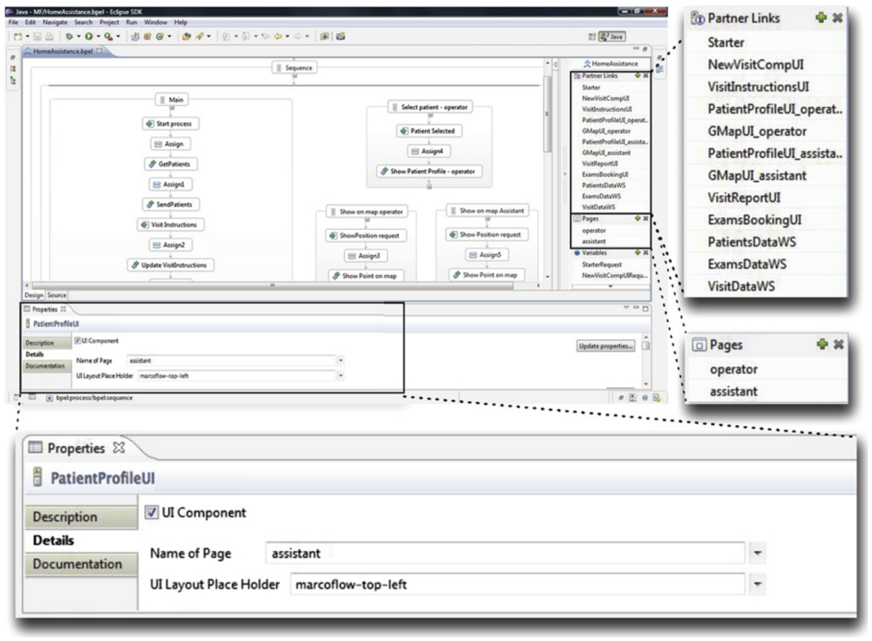
The screenshot in Fig. 11 shows the editor at work. The layout structure of the editor is the same of the standard Eclipse editor, except for some differences in the right and bottom side. On the right side, now it is also possible to define the pages of the UI orchestration (as elements of the *Pages* group). Selecting a page in the list shows the respective details in the *Properties* panel in the lower part of the figure and allows the developer to assign the actor, i.e., the user that will be allowed to access the page, and the HTML template for the page. Still on the right side, where usually there are only partner links for web services, now it is also possible to define UI partner links for UI components. Selecting a partner link from the list again shows its details in the *Properties* panel. Ticking the *UI component* checkbox turns the partner link into a UI partner link and allows the developer to define in which page and place holder inside the page the UI component will be rendered. The actual composition logic is specified in the modeling canvas in the central part of the editor.

The *web-based management console* helps (i) developers deploy ready UI orchestrations and (ii) users in instantiating and participating in running UI orchestrations. Deploying a new UI orchestration requires the developer to pack all the project files (web service WSDLs, UI component WSDL4UIs, BPEL4UI specification, HTML templates, and the system configuration) into a single archive file and to upload it to the management console. Doing so allows the developer to deploy the application by means of a simple mouse click, which invokes the BPEL4UI compiler and generates the standard BPEL file, the event buffers and event proxies, their respective WSDL files, and the UI compositions and then deploys all generated artifacts in the respective runtime environments.

Fig. 12, instead, shows the interface of the management console for regular users, where they can see which UI orchestrations have been deployed they have also access to. Specifically, a user can either start a new instance of UI orchestration (via the upper list in the figure) or participate in an already running instance of UI orchestration (via the lower list in the figure), which – in the case of the operator and assistant in our example scenario – leads him/her, for example, to one of the pages in Fig. 1. The operator is allowed to instantiate the orchestration, and the assistant is enabled to participate.

The MarcoFlow system shown in Fig. 2 is fully implemented and running (a demo of the tool is available at http://mashart.org/marcoflow/demo.htm). In our test setting, we run the UI engine server and the BPEL engine on the same machine, yet these components could also easily be distributed over different physical machines, a feature that is already supported by our code generator.

Developing the MarcoFlow platform in a way that is fully functioning required taking some decisions on the *technologies* to be used. As shown in this paper, we opted for BPEL as service orchestration engine, since BPEL natively supports communication with SOAP/WSDL web services, a requirement that stems from our scenario. We opted for JavaScript UI components, as this represents the current trend in mashups and web-based UI development. Yet, the contributions of this paper are independent of

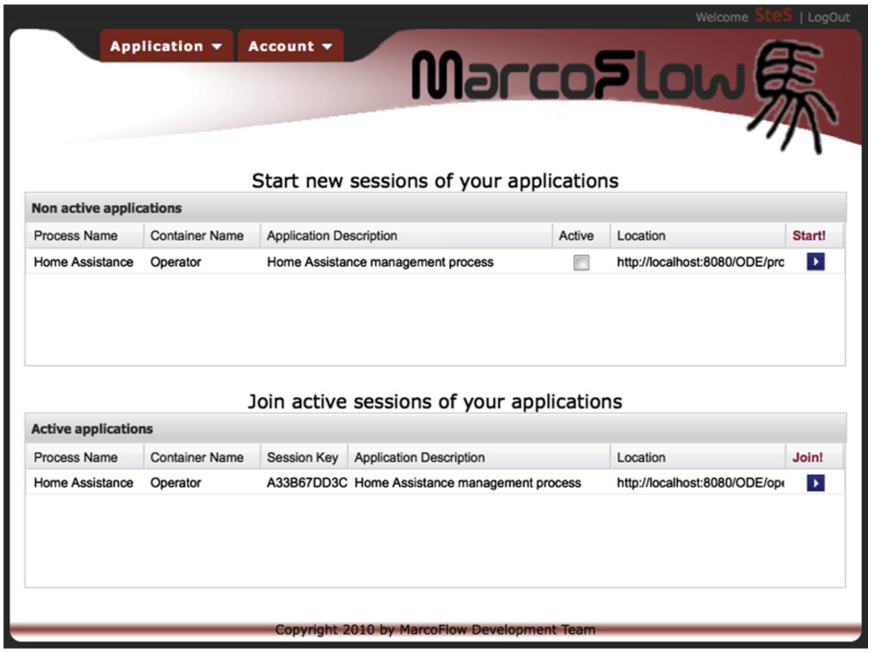


☑ UI Component

Name of Page: assistant

UI Layout Place Holder: marcoflow-top-left

Fig. 11. The extended Eclipse BPEL editor for developing UI orchestrations at work.



Start new sessions of your applications

Non active applications

Process Name	Container Name	Application Description	Active	Location	Start!
Home Assistance	Operator	Home Assistance management process	☐	http://localhost:8080/ODE/prc	▶

Join active sessions of your applications

Active applications

Process Name	Container Name	Session Key	Application Description	Location	Join!
Home Assistance	Operator	A33B67DD3C	Home Assistance management process	http://localhost:8080/ODE/op	▶

Fig. 12. The management console for developers and users allowing them to deploy, instantiate, and participate in UI orchestrations.

these choices and more conceptual than technological (cf. Section 7). In fact, we can easily imagine substituting the BPEL editor with a BPMN editor, of course adding the necessary UI-specific extensions to it. Given the standardized mapping from BPMN 2.0 to BPEL, this would not affect the runtime part of the architecture. If we substitute the BPEL engine with another workflow or business process engine (provided that such already supports interaction with web services), this would require a change in the runtime architecture and the generated process model. But it would be straightforward and not change the philosophy of the overall platform. Similarly, if we want to manage UI integration at the server-side (e.g., via server-side scripting languages like Perl or PHP, ASP.Net or JSP), this could be achieved, but for the cost of lower performance. User interaction occurs at the client side and, hence, UI events are generated inside the client browser. Using server-side technologies means going through the server each time we have a simple intra-page UI synchronization, which degrades the overall user experience. It could however be possible to use different client-side UI componentization technologies, such as W3C widgets (again based on JavaScript), for which we are already studying suitable mashup models [29].

9. Lessons learned

We conclude the paper with a few considerations on lessons learned while developing and applying MarcoFlow.

One observation is that developers seem to prefer a *web-based* environment rather than an Eclipse-based one. We had chosen Eclipse because it already comes with an open-source editor for BPEL, and we felt it was rather powerful and reasonably easy to extend as opposed to developing a new editor. In the end, working with the editor took a lot of time, so that we did not get the benefits of a web-based editor nor the time savings we hoped for.

A second issue relates to the number of *conversions* of messages from SOAP to REST and vice versa. In the current approach, even when two REST services are communicating we always need to SOAP-ify them. While we aim to minimize this kind of conversions as much as possible (by keeping intra-page UI synchronizations on the client), this limits the scalability if a single UI engine is used.

A limitation of the current implementation is that our notification handlers inside the client browser continuously *poll* the server-side event buffers for updates, which further produces communication overhead and possibly delays the forwarding of events. With the growing support for HTML 5 web sockets, we will approach this limitation by pushing events from the server to the client.

Another limitation is the hard-coded assignment of *users* to pages. In our future work we will address this by investigating how resource managers known from workflow management systems can be adapted to our needs. Instead of assigning concrete users, we will therefore assign users roles to pages, which can then be instantiated either at deployment time or runtime.

An interesting finding we did not realize in the beginning is that, since UI orchestrations intermix *stateless* elements (web service invocations) with *stateful* elements (UI components) the need for correlation in UI orchestrations is higher than in pure web service orchestrations. Design-time and runtime constructs here may be needed to simplify specifications and make the engine more scalable.

However the main considerations that will drive our research are in terms of usability and applicability. While working with BPEL was a strong requirement initially, many companies are increasingly considering mashup languages for nonmission-critical applications, targeting relatively simple ways to integrate and present web-accessible data. This would fit well with the MarcoFlow approach, which can be extended to deal with mashup languages.

Finally, working with MarcoFlow and experimenting its usage helped us strengthen our belief that BPEL, its variations, and actually even mashup languages are not suitable for end users, no matter how good development tools are. Our conclusion here is that if we want to bring development power to the end users or at least to knowledge workers we need to define domain-specific models and tools rather than general purpose ones. This is the road we begun to undertake in our efforts within the Omelette EU FP7 project. Yet, we also recognize that UI orchestrations are intrinsically complex, an observation that already inspired a critical survey paper on “process mashups” [30], in which we conclude that the kind of development scenarios supported by MarcoFlow hardly suits the capabilities of less-skilled developers or end users.

In summary, we are confident that the technological limitations of MarcoFlow (no web-based editor, message conversations, polling, user assignments) can easily be addressed in our future work. The conceptual limitations, that is, the intrinsic complexity of UI orchestrations, however, we cannot eliminate.

10. Conclusion

The spectrum of applications whose design intrinsically depends on a structured flow of activities, tasks or capabilities is large, but current workflow or business process management software is not able to cater for all of them. Especially lightweight, component-based applications or Web 2.0 based, mashup-like applications typically do not justify the investment in complex process support systems, either because their user basis is too small or because there is a need only for few, simple applications. Yet, these applications too demand for abstractions and tools that are able to speed up their development, especially in the context of the Web with its fast development cycles.

We introduced an approach to what we call *distributed UI orchestration*, a component-based development technique that introduces a new first-class concept into the workflow management and service composition world, i.e., UIs, and that fits the needs of many of today’s web applications. We proposed a model for UI components and showed how dealing with them requires extending the expressive power of a standard service composition language, such as BPEL. We equipped the language with a modeling environment and a code generator able to produce artifacts that can be executed straightaway by our runtime environment, which separates intra-page UI synchronization from distributed UI synchronization and service orchestration. The result is an approach to distributed UI orchestration that is comprehensive and free.

A strong point of the described approach is that it recognizes the need for abstraction and more expressive models and languages at design time, while – thanks to its strong separation of concerns and powerful code generator – it does not require any new language or system at runtime.

While the intrinsic complexity of UI orchestrations prevents the adoption of MarcoFlow by less skilled developers or end users (which was never the goal of the project), MarcoFlow does provide skilled developers with more expressive power compared to their current instruments: the experienced *BPEL developer* is able to integrate UIs and people into his service compositions; the *mashup developer* is able to design mashups that also involve long-running service orchestrations and user collaborations.

References

[1] B.A. Myers, M.B. Rosson, User interface programming survey, SIGCHI Bulletin 23 (1991) 27–30.
[2] J. Yu, B. Benatallah, F. Casati, F. Daniel, Understanding mashup development, IEEE Internet Computing 12 (2008) 44–52.
[3] F. Daniel, S. Soi, S. Tranquillini, F. Casati, C. Heng, L. Yan, From People to Services to UI: Distributed Orchestration of User Interfaces, in: BPM'10, 2010, pp. 310–326.
[4] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web Services Description Language (WSDL) 1.1, W3C Note, W3C, 〈http://www.w3.org/TR/wsdl〉, 2001.
[5] OASIS, Web Services Business Process Execution Language Version 2.0, Technical Report, 〈http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html〉, 2007.
[6] A. Hofstede, W. van der Aalst, M. Adams, N. Russell, Modern Business Process Automation: YAWL and Its Support Environment, Springer, 2009.
[7] O. Chun, M. La Rosa, A. ter Hofstede, M. Dumas, K. Shortland, Toward web-scale workflows for film production, IEEE Internet Computing 12 (2008) 53–61.
[8] M. Brambilla, S. Butti, P. Fraternali, Webratio bpm: a tool for designing and deploying business processes on the web, in: ICWE, Springer, 2010, pp. 415–429.
[9] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, M. Matera, Designing Data-Intensive Web Applications, Morgan Kauffmann, 2002.
[10] C. Pautasso, BPEL for REST, in: BPM'08, pp. 278–293.
[11] T.v. Lessen, F. Leymann, R. Mietzner, J. Nitzsche, D. Schleicher, A management framework for WS-BPEL, in: ECOWS'08, IEEE, 2008, pp. 187–196.
[12] E.M. Maximilien, A. Ranabahu, K. Gomadam, An online platform for web APIs and service mashups, IEEE Internet Computing 12 (2008) 32–43.
[13] Active Endpoints, Adobe, BEA, IBM, Oracle, SAP, WS-BPEL Extension for People (BPEL4People) Version 1.0, Technical Report, 2007.
[14] Active Endpoints, Adobe, BEA, IBM, Oracle, SAP, Web Services Human Task (WS-HumanTask) Version 1.0, Technical Report, 2007.
[15] R. Acerbis, A. Bongio, M. Brambilla, S. Butti, S. Ceri, P. Fraternali, Web Applications Design and Development with WebML and WebRatio 5.0, in: Objects, Components, Models and Patterns, vol. 11, *LNBIP*, Springer, 2008, pp. 392–411.
[16] J. Gómez, A. Bia, A. Parraga, Tool support for model-driven development of web applications, in: WISE'05, vol. 3806, Lecture Notes in Computer Sciences, Springer, 2005, pp. 721–730.
[17] R. Vdovjak, F. Frasincar, G.-J. Houben, P. Barna, Engineering semantic web information systems in hera, Journal of Web Engineering 2 (2003) 3–26.
[18] D. Schwabe, G. Rossi, S.D.J. Barbosa, Systematic hypermedia application design with OOHDM, in: HYPERTEXT'96, ACM Press, 1996, pp. 116–128.
[19] N. Koch, A. Kraus, R. Hennicker, The Authoring Process of the UML-based Web Engineering Approach, in: IWWOST'01, 2001.
[20] I. Manolescu, M. Brambilla, S. Ceri, S. Comai, P. Fraternali, Model-driven design and deployment of service-enabled web applications, ACM Transactions on Internet Technology 5 (2005) 439–479.
[21] M. Brambilla, S. Ceri, P. Fraternali, I. Manolescu, Process modeling in Web applications, ACM Transactions on Software Engineering and Methodology 15 (2006) 360–409.
[22] Sun Microsystems, JSR-000168 Portlet Specification, Technical Report, 〈http://jcp.org/aboutJava/communityprocess/final/jsr168/〉, 2003.
[23] OASIS, Web Services for Remote Portlets, Technical Report, 〈www.oasis-open.org/committees/wsrp〉, 2003.
[24] F. Daniel, F. Casati, B. Benatallah, M.-C. Shan, Hosted universal composition: models, languages and infrastructure in mashArt, in: ER'09, Springer, 2009, pp. 428–443.
[25] S. Pietschmann, M. Voigt, A. Rümpel, K. Meißner, CRUISe: composition of rich user interface services, in: ICWE'09, Springer, 2009, pp. 473–476.
[26] M. Feldmann, T. Nestler, K. Muthmann, U. Jugel, G. Hübsch, A. Schill, Overview of an end-user enabled model-driven development approach for interactive applications based on annotated services, in: WEWST'09, ACM, 2009, pp. 19–28.
[27] A. D'Ambrogio, A model-driven WSDL extension for describing the qos of web services, in: ICWS'06, 2006, pp. 789–796.
[28] WSPER.org, WS-BPEL 2.0 Metamodel, Technical Report, 〈http://www.ebpml.org/wsper/wsper/ws-bpel20.html〉, 2007.
[29] S. Wilson, F. Daniel, U. Jugel, S. Soi, Orchestrated User Interface Mashups Using W3C Widgets, in: ComposableWeb'11 (ICWE 2011 Workshop Proceedings), Springer, 2011.
[30] F. Daniel, A. Koschmider, T. Nestler, M. Roy, A. Namoun, Toward process mashups: key ingredients and open research challenges in: Mashups'10, ACM, 2010.