From People to Services to UI: Distributed Orchestration of User Interfaces

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

Chang Heng, Li Yan
Huawei Technologies, Shenzhen, P.R. China
{changheng,liyanmr}@huawei.com

Abstract. Traditionally, workflow management systems aim at alleviating people's burden of coordinating repetitive business procedures, i.e., they coordinate *people*. Web service orchestration approaches, instead, coordinate pieces of software (the *web services*), hiding the human aspects that are intrinsically present in any business process behind the services. The recent emergence of technologies like BPEL4People and WS-HumanTask, which introduce human actors into service compositions, manifest that taking into account the people involved in business processes is however important. Yet, none of these approaches allow one to also develop the *user interfaces* (UIs) the users need to concretely participate in a business process.

With this paper, we want to go one step beyond state-of-the-art workflow management and service composition and propose an original model, language and running system for the composition of distributed UIs, an approach that allows us to bring together UIs, web services and people in a single orchestration logic and tool. To demonstrate the effectiveness of the idea, we apply the approach to a real-world home assistance scenario.

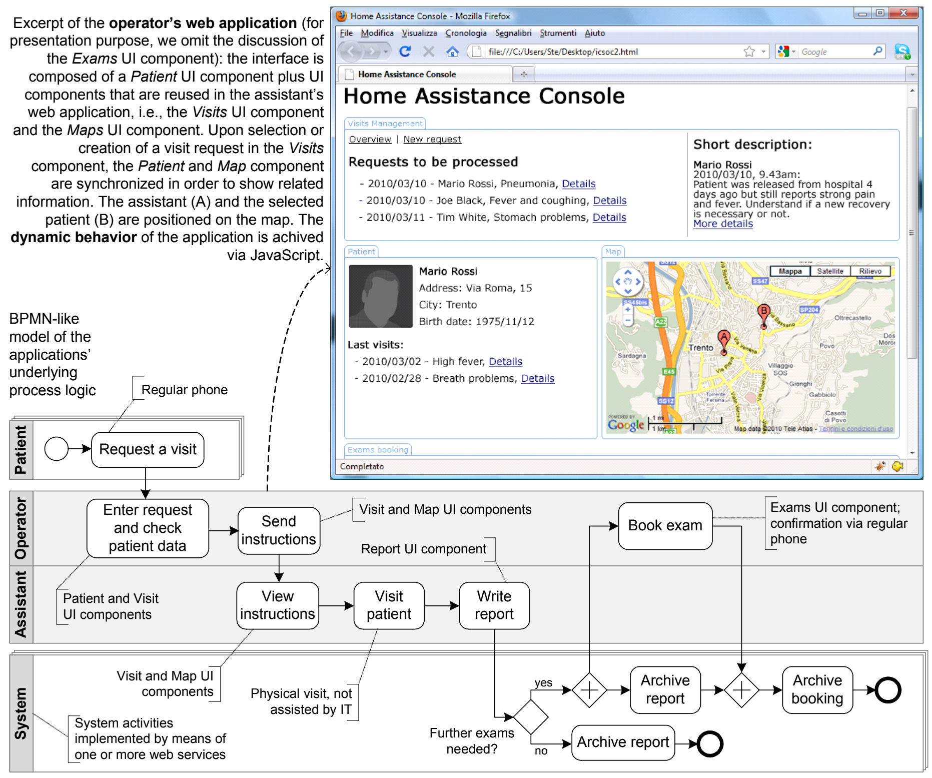
1 Introduction

Workflow management systems support office automation processes, including the automatic generation of form-based user interfaces (UIs) for executing the 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 a possibly complex and sophisticated UI that is tailored to help them in performing the specific job they need 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 is a common need that today is typically fulfilled by developing UIs in ad hoc ways and using a process engine in the back-end for process automation. As an example, consider the following scenario.


Excerpt of the **operator's web application** (for presentation purpose, we omit the discussion of the *Exams* UI component): the interface is composed of a *Patient* UI component plus UI components that are reused in the assistant's web application, i.e., the *Visits* UI component and the *Maps* UI component. Upon selection or creation of a visit request in the *Visits* component, the *Patient* and *Map* component are synchronized in order to show related information. The assistant (A) and the selected patient (B) are positioned on the map. The **dynamic behavior** of the application is achived via JavaScript.

Home Assistance Console - Mozilla Firefox

File Modifica Visualizza Cronologia Segnalibri Strumenti Aiuto

file:///C:/Users/Ste/Desktop/icsoc2.html

Home Assistance Console

Home Assistance Console

Visits Management

Overview | New request

Requests to be processed

- 2010/03/10 - Mario Rossi, Pneumonia, Details
- 2010/03/10 - Joe Black, Fever and coughing, Details
- 2010/03/11 - Tim White, Stomach problems, Details

Short description:

Mario Rossi
2010/03/10, 9.43am:
Patient was released from hospital 4 days ago but still reports strong pain and fever. Understand if a new recovery is necessary or not.
More details

Patient

Mario Rossi
Address: Via Roma, 15
City: Trento
Birth date: 1975/11/12

Last visits:
- 2010/03/02 - High fever, Details
- 2010/02/28 - Breath problems, Details

Map

Map data ©2010 Tele Atlas - Termini e condizioni d'uso

Exams booking

Completato

Google
Mappa
Satellite
Rilievo
Trento
Sardagna
Oltrecastello
Povo
Villaggio SOS
Gionghi
Gabbiolo
Casotti di Povo
BPMN-like model of the applications' underlying process logic
Regular phone
Patient
Request a visit
Operator
Enter request and check patient data
Send instructions
Visit and Map UI components
Report UI component
Book exam
Exams UI component; confirmation via regular phone
Assistant
Patient and Visit UI components
View instructions
Visit patient
Write report
System
Visit and Map UI components
Physical visit, not assisted by IT
System activities implemented by means of one or more web services
yes
no
Further exams needed?
Archive report
Archive booking
Archive report

Figure 1 Simplified home assistance process: gray shaded swim lanes are instantiated only once (in form of suitable UIs) and handle multiple instances of white shaded swim lanes.

Scenario. Figure 1 shows the high-level model of a home assistance process in the Province of Trento 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. There is always one assistant on duty. The *home assistant* views the description, visits the patient, and files a report about the provided service. The report is processed by the *back-end system* and archived if no further exams are needed. If exams are instead needed, the operator books the exam in the local hospital asking confirmation to the patient (again via phone); in parallel, the system archives the report. Upon confirmation of the exam booking, the system also archives the booking, which terminates the responsibility of the home assistance service.

Our goal is to develop an application that supports this process. This application 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 and need to interact with (and are affected by) the evolution of the process. Furthermore, the UI can be itself component-based and created by reusing and combining existing UI components. The two applications, once instantiated, should be able to manage multiple requests for assistance, while the system activities will be instantiated independently for each report to be processed.

Challenges and contributions. 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 (i) understanding how to *componentize UIs and compose them into web applications*, (ii) defining a logic that is able to *orchestrate both UIs and web services*, (iii) providing a language and tool for *specifying distributed UI compositions*, and (iv) developing a runtime environment that is able to *execute distributed UI and service compositions*.

Structure of the paper. Implementing the process of the scenario is a non-trivial composition problem. After describing the UI orchestration approach (Section 3), in this paper we show how defining a new type of binding allows us to leverage the standard WSDL [4] language to describe HTML/JavaScript UI components (Section 4). We then build on existing composition languages (in particular WS-BPEL [5]) to introduce the notions of UI components, pages, and actors to support the specification of distributed UI compositions (Section 5). The extended BPEL is compiled to generate the UI composition logic (that runs entirely on the browser, for performance reasons) and the server-side logic that performs service orchestration and distributed UI synchronization. Finally, we extend the Eclipse BPEL editor to support this extension, and we describe a system that is able to execute distributed UI compositions, starting from the extended BPEL specification. These models and tools are integrated in a hosted development and execution platform, called MarcoFlow (Section 6), jointly developed by Huawei Technologies and the University of Trento.

2 State of the Art in Orchestrating Services, People and UIs

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 invocation 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 [2] 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.alpha-works.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 [3] 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 [17] 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 in this section supports the distribution of UIs over multiple browsers.

3 Distributed User Interface Orchestration: Approach

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 *Visit* 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 it 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 have seen that many of the research challenges raised by the home assistance application are not yet covered adequately by existing works. Especially the aim of providing a single development approach that is able to cover all development aspects in an integrated fashion poses requirements to the *whole life cycle* of UI orchestrations, especially in terms of design, deployment and execution support.

Indeed, supporting the *design* of distributed UI orchestrations such as the ones needed in the example scenario requires:

- Defining a new type of component, the **UI component**, which is able to modularize pieces of UI and to abstract their external interfaces in a way that conforms to the standard WSDL [4] format for service descriptions (to keep compatibility with the BPEL editors and language). We deal with the novel technological aspects introduced by UI components by defining a new type of WSDL binding, which allows us to specify how to translate the abstract WSDL operation descriptions into JavaScript function calls.
- 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 language in order to support UIs (BPEL comes with graphical editors and ready, off-the-shelf runtime engines that we want to reuse, not re-implement). 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. We achieve this by extending the Eclipse BPEL editor with UI-specific modeling constructs that are able to 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 language (UICL), 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.

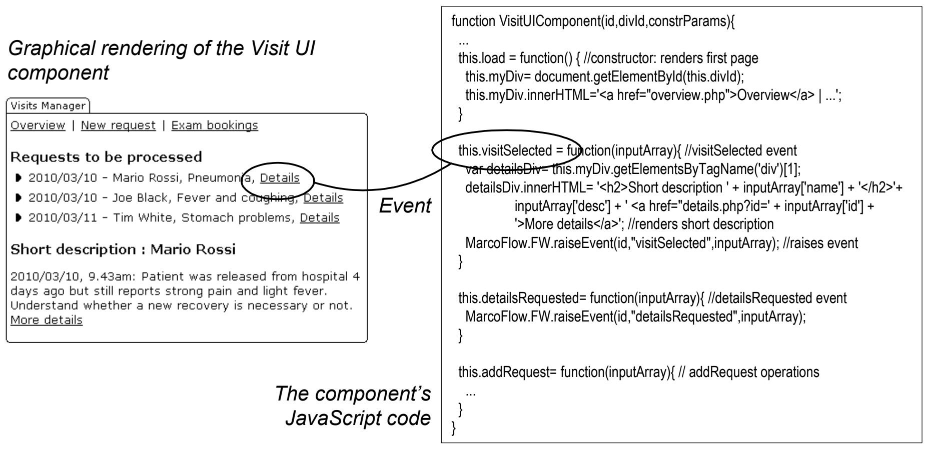
Supporting the *execution* of UI orchestrations requires:

- 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, starting from a UICL specification. Events of a UI component may be propagated to components running in the same web page or in other pages of the application and 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** that is able to run standard BPEL process specifications. The engine is in charge of orchestrating web services and distributed UI-UI communications. We rely on standard technology and reuse an existing BPEL engine.

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*, (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).

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. For the integration of application logic, we rely on standard web service technologies, such as WSDL-SOAP 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.



Graphical rendering of the Visit UI component
Event
The component's JavaScript code

Visits Manager

Overview | New request | Exam bookings

Requests to be processed

- 2010/03/10 - Mario Rossi, Pneumonia, Details
- 2010/03/10 - Joe Black, Fever and coughing, Details
- 2010/03/11 - Tim White, Stomach problems, Details

Short description : Mario Rossi

2010/03/10, 9.43am: Patient was released from hospital 4 days ago but still reports strong pain and light fever. Understand whether a new recovery is necessary or not.
More details

```
function VisitUIComponent(id,divId,constrParams){
  ...
  this.load = function() { //constructor: renders first page
    this.myDiv= document.getElementById(this.divId);
    this.myDiv.innerHTML='<a href="overview.php">Overview</a> | ...';
  }

  this.visitSelected = function(inputArray){ //visitSelected event
    var detailsDiv= this.myDiv.getElementsByTagName('div')[1];
    detailsDiv.innerHTML= '<h2>Short description ' + inputArray['name'] + '</h2>'+
              inputArray['desc'] + ' <a href="details.php?id=' + inputArray['id'] +
              '>More details</a>'; //renders short description
    MarcoFlow.FW.raiseEvent(id,"visitSelected",inputArray); //raises event
  }

  this.detailsRequested= function(inputArray){ //detailsRequested event
    MarcoFlow.FW.raiseEvent(id,"detailsRequested",inputArray);
  }

  this.addRequest= function(inputArray){ // addRequest operations
    ...
  }
}
```

Figure 2 Graphical rendering and internal logic of a JavaScript/HTML UI component

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. Figure 2 shows an example of UI component (the *Visit* UI component of our reference scenario), along with an excerpt of its JavaScript code. 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, e.g., the short description in Figure 2. 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 *Details* link in Figure 2 launches a *visitSelected* 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).
– **Properties**. The graphical setup of a component may require the setting of *constructor parameters*, e.g., to align background colors or to specify the start page of a component.

In order to make UI components available in BPEL, each component is equipped with a standard WSDL descriptor that describes the events and operations (the constructor is expressed as operation) in terms of one-way and notification WSDL operations, respectively. To support the instantiation and execution of components, we have defined a new *JavaScript binding* for WSDL, which binds the abstract operations to the JavaScript functions of the component. The WSDL-UI descriptor can be used as is by the client-side runtime framework and adapted for its use by the BPEL engine.

5 Modeling UI Orchestrations

Specifying 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 BPEL requires extending the expressive power of the language with UI-specific constructs.

5.1 BPEL4UI: concepts and syntax

Figure 3 shows the simplified meta-model of BPEL4UI. 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 in [20]).

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 Figure 3, are:

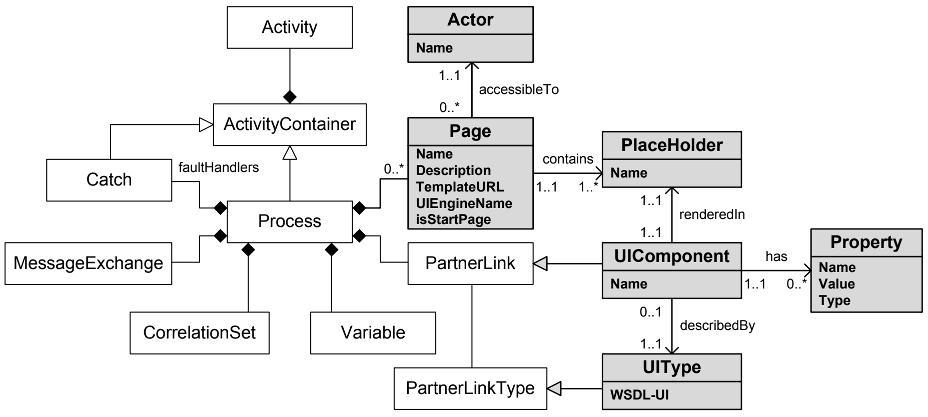


Figure 3 Simplified BPEL4UI meta-model in UML. White classes correspond to standard BPEL constructs [20]; gray classes correspond to constructs for UI and user management.

– **UI type**: The use of UI components in 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 WSDL-UI 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. Also, it is necessary to mark UI component types as such, in order to support the generation of standard BPEL, as described in Section 6.
  As exemplified in Figure 4, we specify the new partner link type like a standard web service type (lines 10-13). In order to reflect the events and operations of the UI component, we distinguish the two roles. Lines 1-8 define the necessary name spaces and import the WSDL-UI 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* (see Section 6) they will run on, and an indication of whether they are a *start page* of the application or not (similar to the start activity in process models).
  The code lines 16-21 in Figure 4 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 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 19-20 in Figure 4.
– **UI component**: UI types can be instantiated as UI components. For instance, there might 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 is part of one process and 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 25-31 in Figure 4 show how to declare the *Visit UI 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 29-30 in Figure 4.
– **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 also dynamically at deployment time or runtime by associating roles instead of actors and using a suitable user management system.
  Actors are added to page definitions by means of the *actorName* attribute, as highlighted in line 18 in Figure 4.

```
 1  <bpel:process name="HomeAssistance"
 2    targetNamespace=www.unitn.it/bpel4ui/HomeAssistance
 3    xmlns:bpel="http://docs.oasis-open.org/wsbpel/2.0/process/
 4    executable" xmlns:visit="http://www.unitnt.it/UI/VisitUIComponent" ...>
 5
 6  <bpel:import namespace="http://www.unitn.it/UI/VisitUIComponent"
 7    location="VisitUI.wsdl" importType="http://schemas.xmlsoap.org/wsdl/">
 8  </bpel:import>
 9  ...
10  <bpel:partnerLinkType name="VisitUIComponent">
11    <bpel:role name="Receive" portType="visit:VisitUI_RECEIVE"/>
12    <bpel:role name="Invoke" portType="visit:VisitUI_INVOKE"/>
13  </bpel:partnerLinkType>
14  ...
15  <pages>
16    <page name="Operator" description="Operator’s home page"
17      templateURL="http://www.unitn.it/BPEL4UI/operatorLayout.html"
18      uiEngineName="UNITN" isStartPage="yes" actorName="Paul">
19        <placeholder name="marcoflow-left"/>
20        <placeholder name="marcoflow-right"/>
21    </page>
22    ...
23  </pages>
24  <bpel:partnerLinks>
25    <bpel:partnerLink name="VisitUI_Operator"
26      partnerLinkType="VisitUIComponent" myRole="Receive"
27      partnerRole="Invoke" isUiComponent="yes" pageName="Patient"
28      placeHolderName="marcoflow-left">
29        <property name="StartPage" type="xsd:string">New Visit</property>
30        <property name="BackgroundColor" type="xsd:string">white</property>
31    </bpel:partnerLink>
32  </bpel:partnerLinks>
33  ...
34  </bpel:process>
```

Figure 4 Excerpt of the BPEL4UI home assistance process (new constructs in bold)

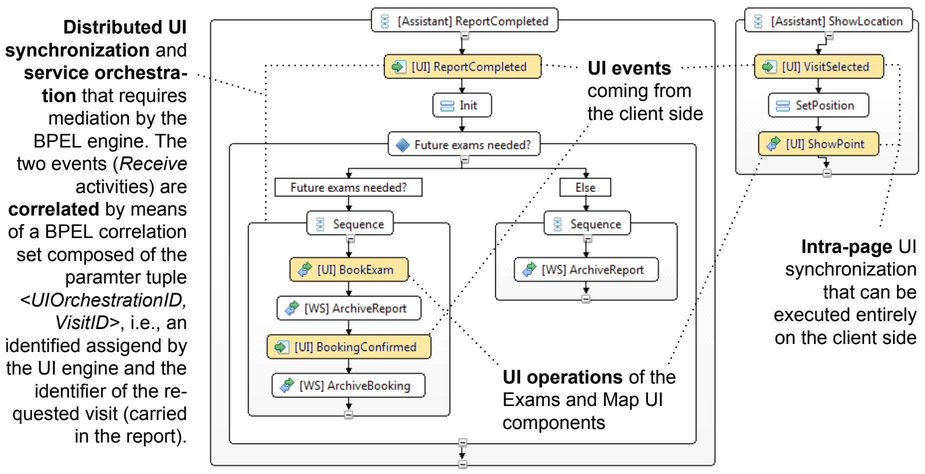

Distributed UI synchronization and service orchestration that requires mediation by the BPEL engine. The two events (Receive activities) are correlated by means of a BPEL correlation set composed of the paramter tuple <UIOrchestrationID, VisitID>, i.e., an identified assigend by the UI engine and the identifier of the requested visit (carried in the report).
[Assistant] ReportCompleted
[UI] ReportCompleted
Init
Future exams needed?
Future exams needed?
Sequence
[UI] BookExam
[WS] ArchiveReport
[UI] BookingConfirmed
[WS] ArchiveBooking
Else
Sequence
[WS] ArchiveReport
UI events coming from the client side
UI operations of the Exams and Map UI components
[Assistant] ShowLocation
[UI] VisitSelected
SetPosition
[UI] ShowPoint
Intra-page UI synchronization that can be executed entirely on the client side

Figure 5 Part of the BPEL4UI model of the home assistance process modeled in the extended Eclipse BPEL editor (these and other *Sequence* constructs run inside a *Flow*).

5.2 Modeling the orchestration logic

The code example in Figure 4 shows that the UI-specific modeling constructs have a very limited impact on the syntax of BPEL and are concerned with the abstract specification of the layout and the declaration of UI partner links. The actual composition logic 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) – it is important to understand the effect individual modeling patterns have on the execution of the final application, i.e., the *semantics* of the patterns. As hinted at in Section 3 and illustrated in Figure 5, we distinguish three main design patterns:

- **Intra-page UI synchronization**: The sequence construct in the right part of Figure 5 shows the internals of the *View instructions* task in Figure 1. When the assistant clicks on a visit request, the patient's address is shown on the Google map. In BPEL terms, we receive a message from the *Visit* UI component (the event) and forward it to the operation of the *Map* component, implementing an intra-page UI synchronization. Both UI components involved in the sequence are associated with the page of the assistant. Hence, this kind of UI synchronization can be performed on the *client side* without involving the BPEL engine.
- **Distributed UI synchronization**: The sequence construct in the left part of the figure, instead, contains a distributed synchronization that cannot be executed on the client only, as the two UI components involved in the communication (*Report* and *Exam*) 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.
- **Service orchestration**: The distributed UI synchronization also involves the orchestration of the *Report archiving* and *Booking archiving* web services, as well as some BPEL flow control constructs. For instance, 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 BPEL4UI excerpt in Figure 5 shows that, when modeling a UI orchestration, it is important to keep in mind who communicates with whom and where UI components will be rendered. 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 so as to turn the intra-page UI synchronization in the right hand side of Figure 5 into a distributed communication and, hence, to require support from the BPEL engine.

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 features provided by BPEL by means of its *Assign* activity. This allows us to leverage on technologies, such as XPath, XQuery, XSLT or Java, for the implementation of also very complex data transformations. Yet, the type of data transformation may affect the logic of the UI orchestration. For instance, if the *SetPosition* activity in Figure 5 does not transform data at all or only performs simple parameter mappings (with the BPEL *Copy* construct) the intra-page UI synchronization can be executed 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 involves exchanging only simple data (e.g., parameter-value pairs) and does not require complex transformations like when interacting with web services. This choice allows us to keep the client-side framework as lightweight as possible, while not giving up any data transformation capabilities. The decision of where to transform data is taken based on the nature of the involved partner links and the type of transformation.

Correlation. The intra-page UI synchronization in Figure 5 does not involve any *asynchronous* communication pattern or multiple entry points into the process logic. It is therefore not necessary to implement any correlation logic in BPEL4UI in order to propagate the *VisitSelected* event to the *ShowPoint* operation. The correlation of the event and the operation in the two web pages is achieved outside the BPEL engine (in the *UI engine server* in Figure 6) by sharing a common key (the *UIOrchestrationID*) that is carried by each event and used to dispatch events. This kind of correlation is automated in our runtime environment and does not require specific modeling.

The distributed UI synchronization, instead, involves two UI events from two different actors: *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 whether they belong to the same process instance or not. In Figure 5, we use *UIOrchestrationID* and *VisitID* (part of the report) as correlation set.

Graphical layout. Defining web pages and associating UI partner links with place holders therein requires implementing suitable HTML templates that are able to host UI components. As we focus on the middleware layer for UI orchestrations, for the layout templates we rely on standard web design instruments and technologies. 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: *<div id="marcoflow-left">... </div>*.

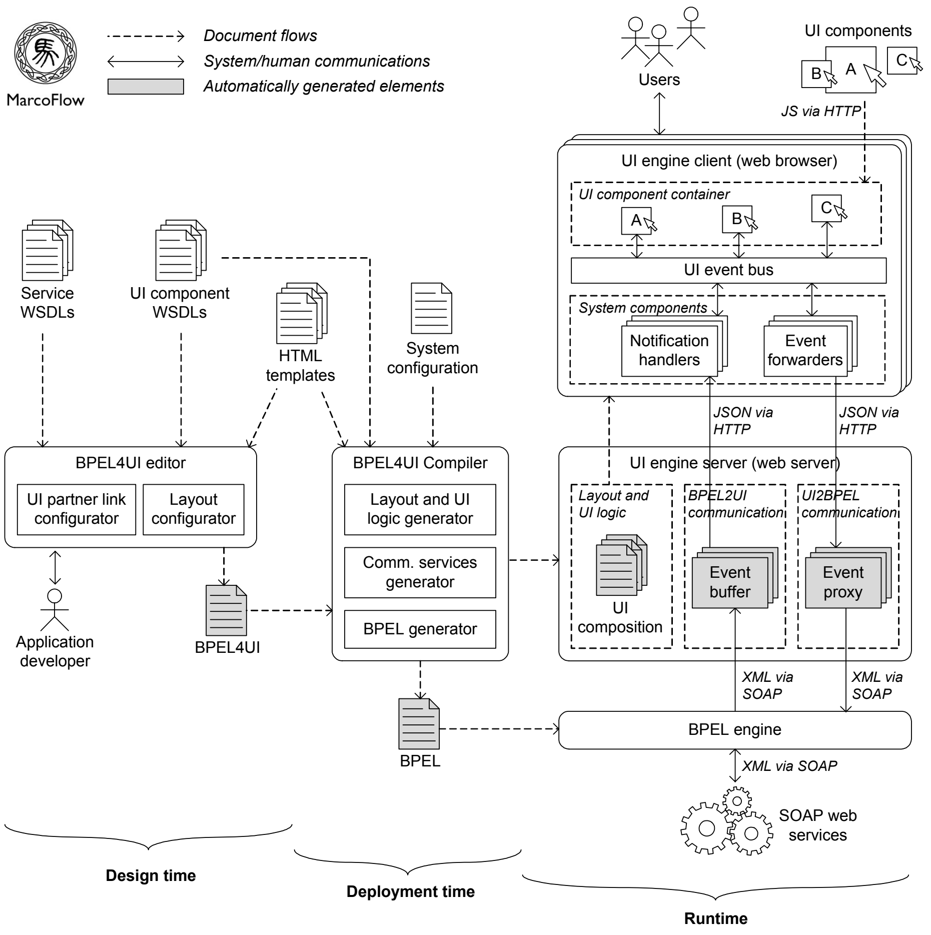


Figure 6 From design time to runtime: overall system architecture of MarcoFlow.

6 Deploying and Running UI Orchestrations

The BPEL4UI language is only a piece of the integrated system for UI orchestration, called *MarcoFlow*. The overall architecture of the system is shown in Figure 6 (for presentation purposes, we discuss a slightly simplified version), which partitions its software components into design time, deployment time, and runtime components.

The **design** part comprises the *BPEL4UI editor* with its *UI partner link configurator* and *layout configurator*. Starting from a set of *web service WSDLs*, *UI component WSDLs*, and *HTML templates* the application developer graphically models the UI orchestration, and the editor generates a corresponding *BPEL4UI* specification in output. The composition logic in Figure 5 has been modeled in our BPEL4UI editor, 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, which is not immediately executable neither by a standard BPEL engine nor by the UI rendering engine (the so-called *UI engine*, which we discuss in the following), into executable formats. This task is achieved by the *BPEL4UI compiler*, which, starting from the *BPEL4UI* specification, the set of used *HTML templates* and *UI component WSDLs*, 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 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 stores them on behalf of the *UI engine client*.
2. A *standard BPEL* specification containing the distributed UI synchronization and web service orchestration logic. Unlike the BPEL4UI specification, the generated BPEL specification does no longer contain any of the UI-specific constructs introduced in Section 5.1 and can therefore be executed by any standards-compliant BPEL engine. This means that all references to UI component partner links in input to the compilation 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* (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, 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 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 publish-subscribe logic ruling the event-to-operation mapping 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* (we are currently studying suitable push mechanisms for events).

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 a web server 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 constant listening. The event buffer is implemented via a simple relational database (in PostgreSQL) 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*,* which is characterized by a simple deployment procedure for BPEL processes.

The MarcoFlow system shown in Figure 6 is fully implemented and running. A demo of the tool is available at *http://mashart.org/marcoflow/demo.htm*.

7 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 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 the idea of *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 their use requires extending the expressive power of standard service composition languages. The language comes with a suitable modeling environment and a code generator able to produce code and instructions that can be executed straightaway by our runtime environment, which separates the problem of intra-page UI synchronization from that of distributed UI synchronization and service orchestration. The result is an approach to distributed UI orchestration that is comprehensive and free.

Unlike in our research on universal composition [3] and unlike mashup tools, in this paper we do not aim at enabling less skilled web users to develop simple applications. MarcoFlow targets skilled web developers that are familiar with BPEL and applications that are complex and possibly involve multiple actors that are distributed over the Web, but that need orchestration. While the idea of event-based UI components has been around for some time now, distributed UI orchestration and multi-browser/multi-actor applications as proposed in this paper are new.

Next, we plan to support the *dynamic selection of actors* (during deployment or at runtime), advanced *access policies*, and *data flow* mechanisms that go beyond the current event-based communication (e.g., through a suitable persistence layer).

References

1. B.A. Myers, M.B. Rosson. Survey on user interface programming. *SIGCHI'92*, pp. 195-202.
2. 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.
3. F. Daniel, F. Casati, B. Benatallah, M.-C. Shan. Hosted Universal Composition: Models, Languages and Infrastructure in mashArt. *ER'09*, pp. 428-443.
4. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note, March 2001. [Online] *http://www.w3.org/TR/wsdl*
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, August 2003. [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.
20. WSPER.org. WS-BPEL 2.0 Metamodel. [Online] *http://www.ebpml.org/wsper/wsper/ws-bpel20.html*