UNIVERSITÀ DEGLI STUDI DI TRENTO
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica
Tesi di Laurea

Marcoflow: extending service orchestration to the presentation layer

Relatore

Prof. Fabio Casati

Laureando

Stefano Tranquillini

Correlatore

Dott. Florian Daniel

Anno Accademico 2008/2009

A mia nipote. Ciao Elena.

Contents

1 Introduction

1.1 Motivation

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.
Only recently, *web mashups* 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.
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 human aspects, which are however intrinsically present in any business process. 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 indeed 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.

1.2 Goal of the work, benefits and results

MarcoFlow project 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. *Scenario example.*

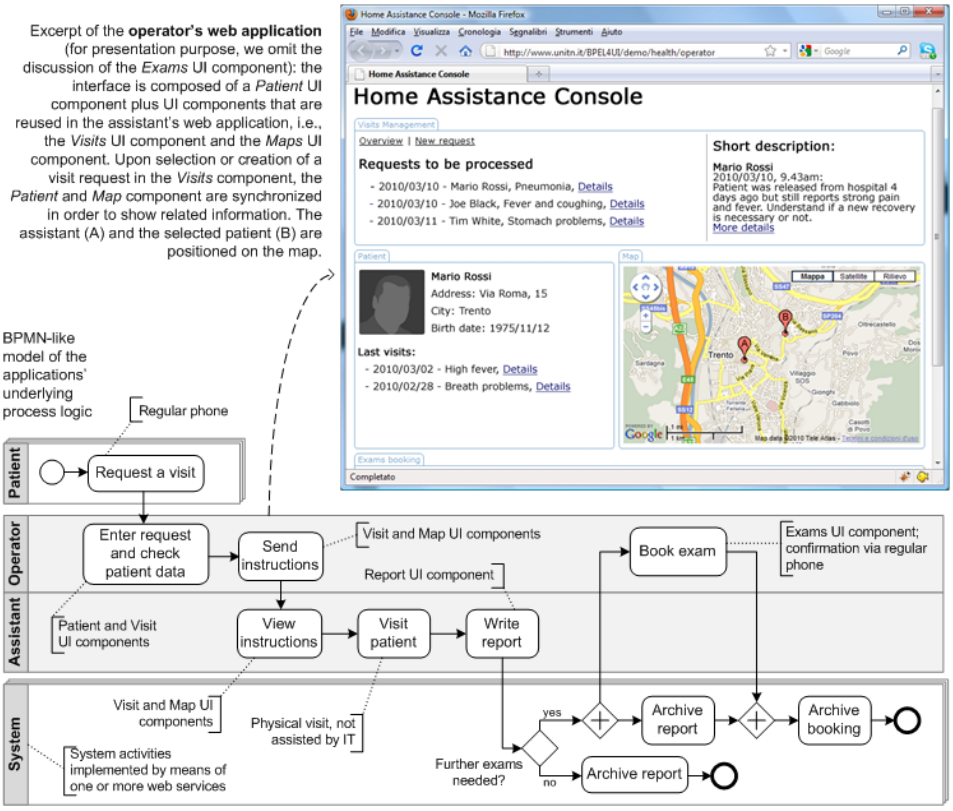

Figure 1.1: Simplified home assistance process: gray shaded swim lanes are instantiated only once (in form of suitable UIs) and handle multiple instances of white shaded swim lanes

Figure 1.1 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 be 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*

We realize this integration in SOA by extending languages and protocols as little as possible and by reusing as much as possible technologies which are BPEL-compliant, so that we use existing BPEL engines and BPEL-related tools, like BPEL design environments. We extent the BPEL editor to facilitate the creation of the BPEL-UI process with the annotation accordingly with the standard annotation of the language.Moreover, this assures the portability of the system, which will work with different execution environments. Finally we implement a prototype runtime environment, which involves the design of models and languages to integrate UI components into BPEL process, in order to execute integrated BPEL process and UI processes.

This solution solves the problem of UI component and service integration in the context of the service-oriented architecture for applications whose user interfaces is distributed over multiple web browsers. The invention covers both the design time aspect and the runtime aspect for such kind of applications by means of an innovative, integrated approach to compose and distribute web services and UI components and a runtime architecture that supports a variety of configurations to execute applications in a distributed fashion and supporting multiple users.

In this prototype we target expert developers with knowledge in service composition. We show how defining a new type of binding allows us to leverage the standard WSDL language to describe HTML/JavaScript UI components, and we define an extension of WS-BPEL, which introduces the concepts of UI components, pages, and actors into the language and supports the specification of distributed UI compositions. 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 our prototype 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:*MarcoFlow*

Contribution of the thesis

The work of this thesis is centred especially on *developing a runtime environment that is able to execute distributed UI and service compositions.* The problems to solve are not only conceptual - model of the components and process - but also regard the technologies involved. The work done aim to find a solution that permits the integration of an orchestration language and components, actually the work was centred to think and find a solution for a middleware that should be able to coordinate the business layer (BPEL process) and components (JS Framework) without effort for developers. In practice the solution covers the *orchestration for distributed UI components and services.*
The benefits of this solution can be found in its expressive power, the possibility to manage UI components among services permits different kind of new solutions in distributed application. First this simplify the work, the developer that has only to think about the behaviour of the application, the coordination of the data is done by the framework automatically. Second the solution allows to have a centralized point where the logic of all users parts are maintained, in this way we can have a simple coordination of the flows and the data. The centralized point is not the only one solution, it can be also a coordination distributed, this means that a web page has com-

ponents with data from different business layers. The solution has widely implication and is something new that, since now, is not reached by other technologies.

Patent and Paper

The MarcoFlow project is not only a prototype for this thesis work. The unique and original UI orchestration approach is being patented in both the United States and the Asia region (the patent application has been submitted). Also, a scientific paper entitled "From People to Services to UI: Distributed Orchestration of User Interfaces" [5] , which describes the work of the whole project (including the work of this thesis), has been submitted to the BPM (Business Process Management) conference 2010, which will be held in September 2010 in New York.

1.3 Reference Scenario

For the first prototype implementation of the MarcoFlow system, we have chosen to focus on an application scenario that is slightly simpler than the one described before (which will be fully implemented for the final demo of the project). The chosen scenario allows us to illustrate in a clear manner all the concepts presented in this work and all its leverages, this scenario shows principally the pragmatically uses that can be done via MarcoFlow. We present a scenario called *FindIT*, which contains UI components and services orchestrated by the process. The scenario's goal is to allow searching locations in the USA trough the Yahoo Locator web service. We build up two versions of this scenario, one with a single user and a second one with two different users, which have different roles in the application. The first one is simple but good enough to see the different kinds of communication (intra page, page to process and process to page). The second implementation is done to show how it is possible to reuse components, that is, multiple instances of the same component. It's also useful to show how the coordination of the messages is managed by the system in a inter-page communication (from page of one user to page of the other one user).

The process model that we are going to present probably will not clearly at the first time, we want to explain better the paradigm that we use. First, the schema has to be read from left to right following the arrows; it contains both components and action involved in the application. The schema is composed by *swim lanes*, and they can be: *gray* or *white*. The gray one describe a UI page, all the things that it contains regards the UI interface of a single page. The white one contains all the actions and component (in this case services) used by the process. In the schema there are two kinds of boxes, the rectangular one is an action performed by the user, components

or process. The rounded rectangle, instead, models UI components; they are physical objects on the UI page of the user. The last item that we can encounter is the *Yahoo Locator*, this is a web services and in this case is used by the process. Now we can proceed with the explanation. In figure

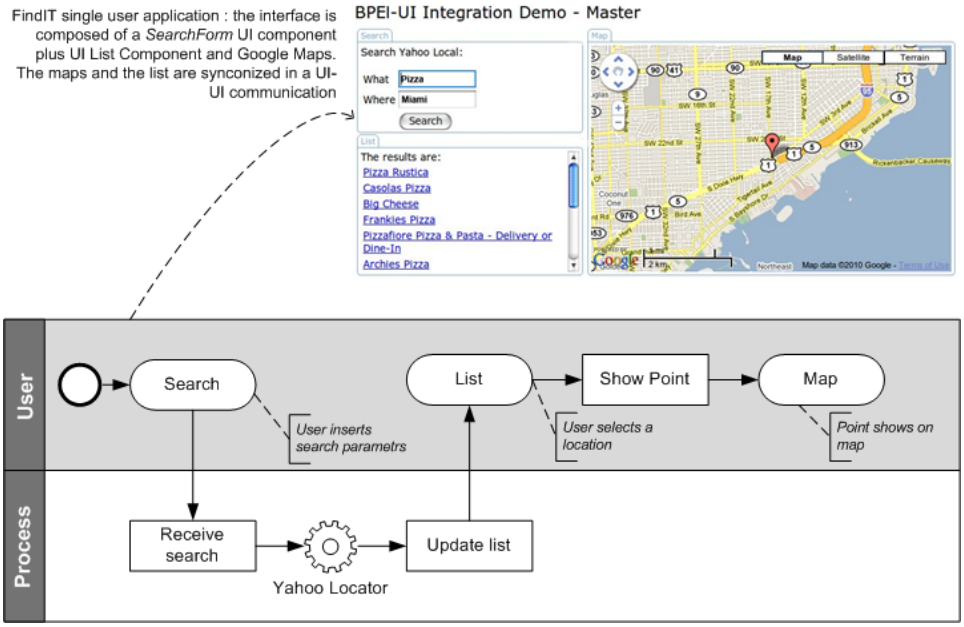

Figure 1.2: Find-it scenario, single user

1.2 shows the high-level model for the Find-It single user application, all components managed in this scenario are:

- *Search form*, a UI component used to send the search request
- *List* UI components used to show the results.
- *Google map* A Google map - a UI component - used to show the marker place of the location selected by the user.
- *Yahoo Locator* a web service that search the locations by some query parameters.

The behaviour of the application is simple: the user inserts the data for the search request, and then, sends at the process a message. The process receive this message, invokes the Yahoo Locator service, which search the locations, and once received the list of location, from the external service, the process update immediately the UI component (list) of the user. At this point, the user has the list of locations on its browser, it can clicks over one of them, in this case a marker place is shown onto the Google map. This scenario is simple, but it contains a communication from components

to process (search), a communication from process to components (update list) and a communication component to component (item selected - show point). All these communication are written in the process, but only the first two need the effort of the process to be executed, the last can be run directly in the browser without effort of middleware and process.

The scenario presented before shows the basic capabilities of this prototype, but this is not enough to understand what the full potentialities are. The scenario shown can be simple recreated by a standard mashup; the next scenario, instead, will introduce some new aspects that are not so easy to do with other technologies. Figure 1.3 shows the high-level model

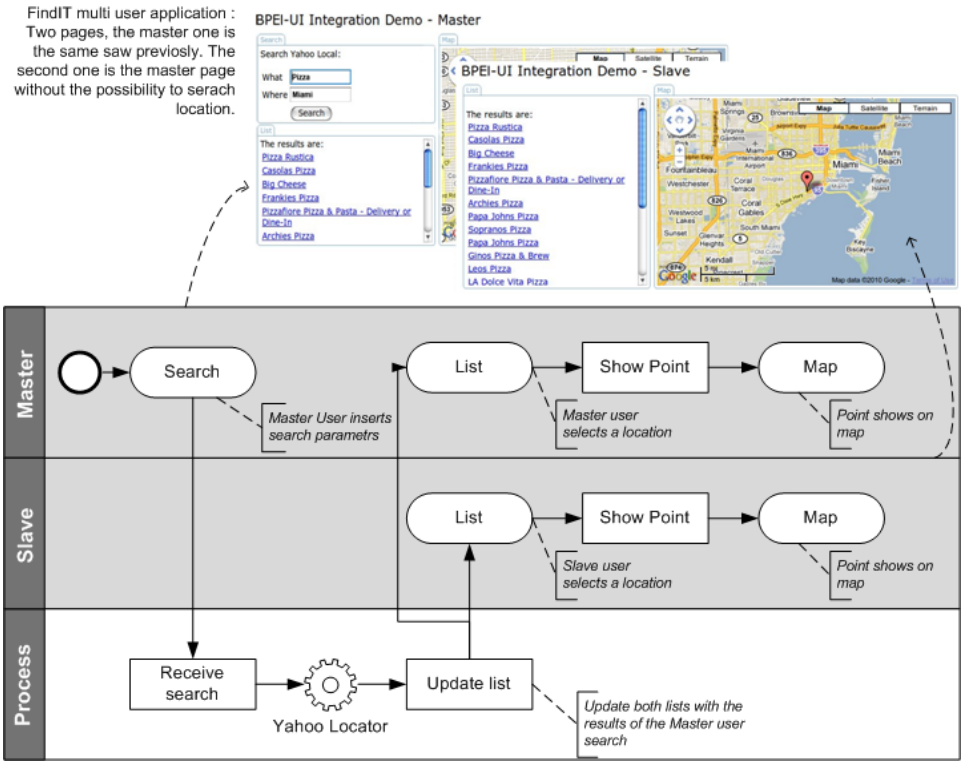

Figure 1.3: Find-it scenario, multiple users

for the Find-It application, but this time for the *multiple* users. Respect the previous one there's another one swim lane for a UI page. In this scenario we introduce two pages, one for the *master* user and one for the *slave* user, with a different behaviour each one. The master page is the same as we saw before, here, there's nothing different. The differences are in the slave page; this page doesn't have the search operation, this means that the user cannot search locations via the web services. The data on the list of this page are provided by the search of the master user, in other words the data found for

the master are forwarded also to the slave page. This concept is new, here we have a synchronization of two different pages, and this is a inter page communication: request from the master page, update to the slave page. The components used in this scenario are:

- *Search form*, a UI component used to send the search request (on the master page).
- *List* UI component that shows on the page the locations founded by the services, two instances: one for the master and one for slave.
- *Google map* A Google map - a UI component - used to show the marker place of the location selected by the user, two instances of it: one for the master one for the slave.
- *Yahoo Locator* a web service that search the locations by some query parameters, used in the process by the master page.

The scenarios presented here are only used as explanation for the project capabilities; the solution proposed in MarcoFlow has it natural placing in the world of the high concurrency and orchestrated application based principally on events. In these cases the prototype shows it potentialities and its powers, in other case it will be much powerful for simple goals. This tool will be use in the applications that are long running and contain high exchanges of user to user communication, for example in the health care environment or in factory processes.

1.4 Structure of the thesis

This paragraph gives a short introduction to the content of every chapter, this want to help the reader to understand the thesis structure. The thesis is constituted by the following seven chapters:

- Chapter 2: Presentation of the state of the art, in this chapter there is a review of the principal technologies and ideas used in this thesis; it's from web services to mashups passing through the workflow management.
- Chapter 3: The whole prototype of MarcoFlow, this chapter will be useful to understand all the projects, from the models to describe the logic of the runtime application passing through the generation of all the artifacts that are in the middleware.
- Chapter 4: The idea proposed for the middleware solution. An introduction about the requirements, the problems to avoid and an explanation about how we solved them.

- Chapter 5: Theory is not all, in this chapter we introduce the creation of the section described in chapter 4, how it works in practice, what data are used during the creation and execution and an explanation that permits to understand why it works.

- Chapter 6: *"A picture is worth a thousand words"*, in this case a demo. Here we explain with an example scenario (presented before) the whole behaviour of the project with a specific focus on the things discussed in this thesis.

- Chapter 7: The state of the solution, and a planning of the future works.

2
State of the art

2.1 Introduction

In this chapter we will discuss about the state of the art over web services and workflows. The web services are the core of the server side of the MarcoFlow's engine, the communications are based on these technologies. The workflows are introduced to explain the coordination and control that is used in our process. The BPEL (Business Business Process Execution Language) models a process like a sort of workflow; a workflow for managing services. Our BPEL-UI language is more closed to the workflow, in fact, inside this language there is a specification about roles and actors that is typically of workflows. Finally we use wf to explain how is possible to use them to manage UI components; in the last decade the mashups becomes widely used in the web, they are a kind of workflow that manages data and user interfaces.

2.2 Web services

"A software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically Web Services Description Language WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards" this is a web service for the W3C. [18]
The web service was introduced to realize distributed environment where all the components can interact without problems. These problems mentioned are usually the interoperability of the different languages and the platform used. Using web services there are some advantages:

- *Interoperability* different application written in different languages and running on different machine can interact without problems.
- *Understandable* the data exchanged are in a textual format, then a message is simple to analyze and understand for the developer.

- *Protocol* the web service uses the HTTP protocol, no new rules for the firewall.
- *Reuse and combine* is simple to reuse old block of application, or integrate different services to create a new integrated service.
- *Maintenance* the layers are separated, the maintenance is simply to do, there is only to change the code of the services.

2.2.1 The Web Services Description Language (WSDL)

WSDL (Web Services Description Language or Web Services Definition Language in version 2.0) is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information [3]. There are two different version of WSDL: 1.1 and 2.0, these have some differences in the skeleton and in the providing functionality. The latest version is released in the 2007 and nowadays is not fully supported in all the systems, respects the previous one it fully supports the HTTP protocol (not only the GET and POST method). This means that also the REST services (explaining after) can be described using the new version of the WSDL document. There are also other features added: the inheritance (an interface can be built using one or more interfaces), a standard way to extend the languages (plus Message Exchange Patterns and Features and Properties based on SOAP 1.2), the safe operation support and the SOAP 1.2 binding. [8]

2.2.2 Simple Object Access Protocol (SOAP)

SOAP (Simple Object Access Protocol) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols[8].
Principally, SOAP protocol defines:

- Message format, how the message can be converted into a XML doc.
- Transport, how the message should be transport using different protocols (like HTTP or SMTP etc).
- A set of rules that have to be followed during the processing of the SOAP message and a classification of the entries involved in the process of SOAP messages. It also defines what parts of the message should be read by whom and how they can do.
- A set of convention about how to turn an RPC call into a SOAP message and vice versa.

Two parts, the header and the body compose an envelope of SOAP message. The header (is an optional field) is usually used for coordination information or additional information about security or trust. SOAP and WSDL languages, together, gives additional features that permits to manage the security of the messages, addressing and other things. The body (is mandatory) contains the message part, or, in other word the information exchanged in the request/response. The body, for the same operation with the same output and input, can be appear different sometimes, this field, contains an encoding of the message and this property defines the schema of the message encoded. In the WSDL-SOAP paradigm there are four types of possible encoding that are: RPC/encoded, RPC/literal, Document/encoded or Document/literal, probably the last one is the most used, especially in the wrapped version. For more information about this encoding, see [2].

2.2.3 RESTful web services

Since now we trade the services as the classical and most known web services, so they are usually have WSDL descriptor and uses SOAP style to exchange the messages. This kind solution is used, but in the last years, with the growing of the lightness web application, RESTful services become used for this purpose.
REST stands for REpresentational State Transfer and was first defined within a PhD thesis by Roy Fielding. REST idea is about using the principles of the World Wide Web to build applications. It works over the HTTP protocol and exploits all it leverages, it follows some architectural principles:

- *Addressable Resources*, all the things on the network has a proper ID, to be specific, this identification is given by the URI

- *Uniform interface* All the services are built over the HTTP protocol, so they following its operations. This means that a RESTful services has GET, POST, PUT and DELETE as possible operations. Usually GET and POST are used to retrieve and send data, PUT to create an instances and DELETE to delete the instances. But these uses of the operations are not mandatory.

- *Representation oriented* They have the specific interface, but the data exchanged with the client can be in different forms. As far we know, different platforms needs different formats (i.e. AJAX wants JSON, JAVA wants XML), the REST paradigm allows having different type of representation for the same service.

- *Communicate stateless* This give a huge scalability, actually they doesn't store the client session on the server, so each service can be fast reusable. In order to perform check over user's session, this information has to be sent by the client to the service.

Each object on the web that follows these claims can be called: RESTful services (it can be a servlet, a jsp, etc). They has the power of lightness, and they offers a simple, interoperable, and flexible way of writing web services that can be very different than the RPC mechanisms like CORBA solution and standard services. RESTful service actually has some weakness, there's no standard definition and there's no solution about transaction reliability, etc. This kind of services is commonly used in AJAX application, where send or retrieve data is the main issues and the security is not a milestone of the application [9]. The latest version of WSDL document permits to describe the RESTFul services by a WSDL file.
These are the most used technologies in web services, but in the history there was different solution proposed to solve the interoperability of applications among the net.

The first solution that we can treat in the web service category was the: **RPC (Remote Procedure Call)**. This is a communication technology that allows calling a procedure from a computer to another, different, one. Usually these two elaborators are connected on the net, and the request for the execution of the subroutine, goes through this connection. For the developer, the code for calling an RPC looks like a normal calls for a standard procedure (local). This idea was born in the 70's and widely used in the 80's, usually these solution are based on the concept of client-server paradigm: The client sends a request to a remote server, this request is made in order to perform a specified procedure. A response is returned to the client which can read and continues its works. These calls are usually blocking, so the program has to wait for the response before proceed. The big disadvantage of this solution is that there's no interoperability between different languages, so the RPC works only if the language are the same both for the server and the client.
There's a variation of RCP for the JAVA languages, is called **JAX-RPC (Java API for XML-based RPC)**. The differences are not only for the program language but also in the type of message used. This solution introduce a XML based message and a WSDL description for the service. These two things give a little bit sensation of standard and usability, but the problem of JAX-RPC remains the same that has already discovered for the RPC: the solution works only for a single language, in this case JAVA language.

A different solution was proposed by *OMG (Object Management Group* in the early 90's named **CORBA (Common Object Request Broker Architecture)** solution. This system uses a proper description language called IDL, this describes the public interface of a service; that file is used inside the framework where a mapping over information of this file and the code is perform. At first side this solution can be appear identical as web service, but there are some differences that distinguish these two solutions.

These dissimilarities are the mainly causes of the lean uses of this solution, and can be researched in the middleware of this system. The CORBA specification wants an ORB component that performs the interaction through different objects; this means that on the server there must be a CORBA Object to allow the interaction. This is one of the things that has block the CORBA diffusion, not all the people wants to install proprietary software on proper server. There are also other facts that cut off CORBA as service communication solution: to facilitate the transport of messages their needs a dedicated protocol. There was different solution proposed, and at the end IIOP was selected as mainly solution. This new protocol was developed in the same period of the HTTP, but the HTTP has more advantages respect the IIOP and so all takes the HTTP as standard for communication.

2.3 Workflow management

Workflow is typically a separation of work activities into well-defined tasks, roles, rules, and procedures which regulate most of the work in manufacturing office. It can be also defines as a collection of tasks organized to accomplish some business process. A task can be performed by one or more software system, one or more team of humans, or a combination of these. [7]. There are several categorization of the wf, one of these [13] categorize process into: (i) *material process*, (ii) *information process* and (iii) *business process*. The first one has a goal to assemble component (physical) and deliver products. The second one (information) is related to automated and partial automated tasks that create, process, manage and provide information. The last one (business) are market-centred description of an organization's activities, implemented with the two previous types of processes. When an organization takes its business in term of business process it can regenerate each process to adapt and improve the changing requirements. Workflow is a concept related to the reengineering and automating of business and information process (we will see after). An interesting characterization of workflow given by Georgakopoulos [7], they characterize them along a continuum form human-oriented to system-oriented. The first one involves human collaborating in performing and coordinating tasks, the second one involves computer system that performs computation-intensive operation and specialized software tasks.

Workflow Management or WFM is a technology used to reengineering of the process, it involves a workflow definition that describe the aspect of a process which is relevant to controlling and coordinating, it also tasks and providing a design and implementation of the process, business needs and information system exchange. They usually permit to have an increment of the process's efficiency, a better control of the process and flexibility

of the work that can be planned in time. A WFM is usually: component oriented, supports workflow application (so process), ensures the correctness and reliability and supports the evolution. They are the software transposition of the workflow idea.
Management of a workflow includes: (i) a process modelling and workflow specification, (ii) a process reengineering and (iii) a workflow implementation and automation. The first one regarding the process, it has to be understood and described in different level of abstraction. Performing a specification requires a workflow model, it typically uses a set of concepts useful to describe the process like role that performs tasks. The specification is typically done using a workflow specification language that uses rules, constraints, and or graphical constructs to describe the order and the synchronization of the tasks. The second step, has as goal the re-engineering of the business process, this is done to optimize the process. The strategy that optimizes the process is given by the final objectives. The last one the (iii) is the issues associated with the realizing of workflow, it includes the uses of computers, software, information system etc.

Workflow model is the modelling of a workflow process, and usually is done by a workflow management system, these technology typically support the activity-based model (there is also communication-based model). The WFMSs most support the activity-based model this consist in a series of elements: workflow, tasks, objects, roles and agent that are common oriented to the activity and not to the communication. They uses are quite obviously, workflow define the order of tasks, tasks defines the order of operation, object defines the item used inside an operation, roles define the skill or the information required to perform a task. Agents, instead, can be human or an information system that fill roles, execute task and interact in the workflow execution. Usually WFMSs provide a graphical workflow design and perform some issues: *task structure* also known as control flow, it can specify that task has to be executed in parallel or in sequence etc, *exception handling* specify what action do when a task fail, *task duration* that is time of a task and *priority* for the task. A WFMS usually have a testing and analysis unit where the bottlenecks can be founded, has also a sort of architecture interoperability: there are some APIs that can be used to interact, the system has also a correctness and reliability mechanism that is used when a workflow has to manage concurrency control or applications failure.
The workflow was born to manage person and jobs, and they do this job very well. A good planning of jobs with task and roles can simplify the work and save time and money. With the growing of the web environment and the uses of services, the workflow was applied on other field: the managing of services.

2.3.1 BPEL: Business Process Execution Languages

This can be seen as the first evolution of the workflow, or better, a new way to use workflow. Since then the workflow was used to manage people and works, with the BPEL language they starts to be used to manage web services. This topic regarding also service composition because this is its goal, uses different WSs and builds up a new service with new functionality.

BPEL (Business Process Execution Languages) is commonly called WS-BPEL [1] and is also a representative of service orchestration approaches. It's a standard composition language by OASIS, provides an XML-based grammar to describe the control logic required to coordinate web services participation in a process flow. WS-BPEL can act both as coordination protocol and proper composition language. WS-BPEL orchestration engines can execute this grammar, coordinate activities and compensate activities when errors occur. More than 37 organizations collaborated to develop WS-BPEL, including representatives of Active Endpoints, Adobe Systems, BEA Systems, Booz Allen Hamilton, EDS, HP, Hitachi, IBM, IONA, Microsoft, NEC, Nortel, Oracle, Red Hat, Rogue Wave, SAP, Sun Microsystems, TIBCO, webMethods, and other members of OASIS. BPEL is based on WSDL-SOAP web services, and BPEL processes are themselves exposed as web services. Control flows are expressed by means of structured activities and may include rather complex exception and transaction support. Data is passed among services via variables (Java style).
So far, BPEL is the most widely accepted service composition language and several extensions have been developed, i.e. BPELlight [14] BPEL4SWS [11], BPEL4PEOPLE [10]. BPELlight allows one to specify process models independent of Web service technology. Since its interaction model is based on plain message exchange, it is completely independent of any interface description language. BPEL for Semantic Web Services (BPEL4SWS) uses Semantic Web Service Frameworks to define a communication channel between two partner services instead of using the partner link which is based on WSDL 1.1. It enables describing activity implementations in a much more flexible manner based on ontological descriptions of service requesters and providers. In January 2008, OASIS issued a Call for Participation in the BPEL4People Technical Committee. The group works to define a WS-BPEL extension enabling the definition of human interactions ("human tasks") as part of a WS-BPEL process. In 2004, W3C defines WS-Choreography Definition Language (WS-CDL)[19], an XML-based language that describes peer-to-peer collaborations of parties by defining, from a global viewpoint, their common and complementary observable behaviour; where ordered message exchanges result in accomplishing a common business goal. Many variations of BPEL have been developed, e.g., aiming at invocation of REST services [16] and at exposing BPEL processes as REST

services [17].
BPEL language is a powerful to orchestrate services, the orchestration means: it can specifies an executable process that involves message exchanges with other systems, such that the message exchange sequences are controlled by the orchestration designer. This is different from the choreography that specifies a protocol for peer-to-peer interactions, defining, e.g., the legal sequences of messages exchanged with the purpose of guaranteeing interoperability. In fact the difference between orchestration and choreography is executions and control.

There were ten original design goals associated with BPEL language [12]:

- *Web Services as the Base*: should define business processes that interact with external entities through Web service operations.
- *XML as the Form*: defines business processes using an XML based language
- *Common set of Core Concepts*: should define a set of Web service orchestration concepts that are meant to be used in common by both the external (abstract) and internal (executable) views of a business process
- *Control Behaviour* should provide both hierarchical and graph-like control regimes, and allow their usage to be blended as seamlessly as possible.
- *Data Handling*: provides limited data manipulation functions that are sufficient for the simple manipulation of data that is needed to define process relevant data and control flow.
- *Properties and Correlation*: should support an identification mechanism for process instances.
- *Long-Running Transaction Model*: should define a long-running transaction model
- *Modularization*: should use Web services as the model for process decomposition and assembly
- *Composition with other Web Services Functionality*: should build on compatible Web services standards and standards proposals as much as possible in modular manner.

This list has a lot of points in common with the idea of workflow, BPEL can see like a workflow for services, so we can argue that BPEL language is an evolution, in other environment, of the workflow idea. However the workflow

is evolved, from managing of people to a managing of services. Now a day the use of workflow is taken into account in another field of study, the UI composition and data integration.

2.3.2 UI composition

Historically, UI composition has first been considered for desktop applications; these applications use general-purpose programming languages to integrate components (e.g. Java). All interactions with the component are performed via the component UI logic. The only way to integrate a component application is to have an intimate knowledge of its UI, be able to track the mouse position and/or to intercept the text entered by the user, and in this way understand what is shown by the UI component and possibly even execute actions that cause UI modifications (e.g., by having the composite application simulate mouse clicks or keyboard strokes). The integration in this case is a daunting task.
Probably there is no UI composition approaches readily available [6]. Desktop UI component technologies such as .NET CAB or Eclipse RCP are highly technology-dependent and not ready for the Web. Browser plug-ins such as Java applets, Microsoft Silverlight, or Macromedia Flash can easily be embedded into HTML pages; communications among different technologies remain however cumbersome (e.g., via custom JavaScript). Java portlets or WSRP represent a mature and Web-friendly solution for the development of portal applications; portlets are however typically executed in an isolated fashion and communication or synchronization with other portlets or web services remains hard. Portals do not provide support for service orchestration logic, and this can be a big limitation.

Portals and portlets

Still in the context of web applications, portals and portlets represent a different approach to the UI integration problem on the Web. Their approach explicitly distinguishes between UI components (the portlets) and composite applications (the portals) and it is probably the most advanced approach to UI composition as of. Portlets are full-fledged, pluggable Web application components that generate document markup fragments and facilitate content aggregation in a portal server. Portlets are conceptually very similar to servlets. The main difference between them consists in the fact that while a servlet generates a complete web page, portlets generates just a piece of page (commonly called fragment) that is designed to be included into a portal page. Hence, while a servlet can be reached through a specific URL, a portlet can only be reached through the URL of the whole portal page. A portlet has no direct communication with the web browser, but this communications are managed by the portal and the portlet container that allow

the request-response flows and the communication between portlets. A portal server typically allows users to customize composite pages and provide single sign-on and role-based personalization.

Web Mashup

Web mashups are web applications that are developed by combining content, presentation, and application functionality from disparate Web sources. The term mashup typically implies easy and fast integration based on open APIs and data sources, yielding applications that add value to the individual components of the application and thereby often use components in ways that differ from the actual reason that led to the original production of the raw sources.
Mashups are strongly related with the Web. The Web is the natural environment for publishing content and services today, and therefore it is the natural environment where to access and reuse them. Content and services are published in a variety of different forms and by using a multiplicity of different technologies.The very innovative aspect of web mashups is that they integrate sources also at the UI layer, not only at the data and application logic layers. Integration at the data and application logic layers has been extensible studied in the past, while integration at all three layers is still a goal that put architects and programmers in front of important conceptual and technical problems.
Mashup development is still an ad-hoc and time-consuming process, requiring advanced programming skills (e.g., wrapping web services, extracting contents from web sites, interpreting third-party JavaScript code, etc). There are a variety of mashup tools available online, but, as we will see, only few of them adequately address the problem of integration at all its layers. In this section, we give an overview of the state of the art in the mashup world, spanning from manual development to semi-assisted and fully assisted development approaches.

There are principally three kinds of mashup tools:

Manual development Developing applications that aggregate data, application logic and UIs coming from diverse sources requires deep knowledge about technologies. In addition, it might be necessary to master the business protocols of employed services and to have knowledge about how to compose services into service orchestrations. Usually for manual mashup development, the user would not only need to be conversant in some scripting language such as JavaScript or some other general purpose language, but also the APIs and data formats used in all the services composed in the mashup application. Therefore the complexity of manual mashup development and maintenance puts usability beyond the scope of an average web

user and limits the scope to only skilled programmers.

Semi-assisted development These solutions are built to speed up and simplify the development of mashed up solution. Typically, they address the problem of data extraction from web sites and the provisioning of such data in form of data services or re-usable user interface elements. There are principally two used application for this proposed:

- Dapper [1] is a free online instrument for the generation of data wrappers that extract data from well-structured web pages. Dapper is based on a point and click technique able to assist the user in the selection of the contents to be extracted and to infer suitable extraction rules. Specifically, data extraction leverages the structure of the HTML formatting to understand which elements to extract

- Openkapow [2] is a similar open service platform based on the concept of extraction robot, that is, user-created wrappers. Users of Openkapow can build their own robots, expose their results via web services, and run them from openkapow.com for free. Robots are able to access web sites and support the extraction and reuse of data, functionality and even pieces of user interfaces. Robots are built through a visual development environment called RoboMaker.

Fully-assisted development Mashup tools or mashup platforms addresses the problem about how simplify the creation of data extraction instrument for web pages, and also the composition of components into application. The solution saw before needs an effort from the developer that probably is too heavy. The user hast to have a widely background over a variety of web technologies. There are different solutions in the web, we will discuss over some of them.

- Yahoo! Pipes [3] Provides a simple and intuitive visual web editor that allows one to design data-centric compositions. It takes data as input and provides data as output; the most important supported formats are RSS/Atom, XML, and JSON. A pipe is a data processing pipeline in which input data (coming from diverse data sources) are processed, manipulated and used as input for other processing steps, until the target transformation is completed. Is the same concept of pipe to the one employed in UNIX in which simple commands are combined together in order to create some desired output .This pipeline-style process is implemented through an arbitrary number of intermediate operators, which manipulate data items inside the data feeds or provide features like loops, regular expressions or more advanced features

[1] "http://www.dapper.net/open/"
[2] "http://openkapow.com"
[3] "http://pipes.yahoo.com/pipes/"

like automatic location extraction or connection to external services. The set of operators are predefined and fixed; new functionality can be included in form of web services. Also, stored pipes can be reused as sources of another pipe.
In order to create an application, a user can drag-and-drop graphical elements on the canvas and wire the elements together thus creating sophisticated expressions for fetching and manipulating data from different sources on the Web. Yahoo! Pipes' development environment is characterized by a simple and intuitive development paradigm that is however targeted at advanced web users or programmers. In fact, the level of abstraction of its operations and the characteristic data flow logic is only hardly understandable to non-programmers. Pipe's output is not meant for human consumption but rather for integration in other applications. This limits both the variety of input sources that can be used and the accessibility of its output. In fact, the absence of any support for UIs prevents the direct use of Pipe's output by common web users. However, Pipes is a very popular data-mashup development tool, very likely due to its efficient and intuitive component placing and connection mechanism.

- Microsoft Popfly [4] This solution (discontinued as of August 24th 2009) gained a great consensus in the mashup community and achieved good levels of popularity and usage. Popfly provides a visual development environment for the realization of mashups based on the concept of components, or block as they are called in Popfly. A composition is created by dragging and dropping blocks of interest onto a design canvas and by graphically connecting them to create the desired application logic. A block can take the role of connector to external services or it can represent some internal functionality (implemented through a JavaScript function). Each block provides input and output ports that enable its connection to other blocks. Blocks can also be used to provide a user interface that can display the result of some processing. Placing multiple visualization blocks into a same page allows one to define the overall layout of the page. The internal layout of blocks can be customized by inserting ad-hoc HTML, CSS or JavaScript code. Popfly has a wide collection of available blocks, offering functionalities like RSS readers, service connectors, map components based on Virtual Earth, etc. New blocks and compositions can be defined saved, shared and managed in a dedicated section of the platform.
  At runtime, the communication flow is event-driven, that is, the activation of a certain component depends on the raising of some event by another component. There is no support for exception and trans-

[4] "http://popflyteam.spaces.live.com-MSPopflyhasbeendiscontinuedsinceAugust24, 2009."

action handling, but Popfly provides a section dedicated to the test and preview of the composition. Ready compositions are stored on the Popfly server, but the execution is done on the client - as many of the built-in blocks are based on the Silverlight platform.

- Intel Mash Maker [5] provides a completely different mashup approach: an environment for the integration of data from annotated source web pages based on a powerful, dedicated browser plug-in for the Firefox or I.E. web browser. Rather than taking input from structured data sources such as RSS/Atom feeds or web services, Mash Maker allows users to reuse entire web pages and, if suitably annotated, to extract data from the pages. That is, the "components" that can be used in Mash Maker are standard web pages. If a page has been annotated in the past, it is possible to extract the annotated data from the page and share it with other components in the browser. If the page has not been annotated, it is possible reuse the page as is without however supporting any inter-page communication.
  In order to annotate a page, Mash Maker allows developers and users to annotate the structure of web pages while browsing and to use such annotations to scrap contents from annotated pages. Annotations are linked to target pages and stored on the Mash Maker server in order to share them with other users.
  Composing mashups with Mash Maker occurs via a copy/paste paradigm, based on two modes of merging contents: whole page merging, where the content of one page is inserted as a header into another page; and item-wise merging, where contents from two pages are combined at row level, based on additional user annotations. The two techniques can be used to merge also more than two pages. Data exchange among components is achieved by means of a blackboard-like approach, where data of components integrated into an application are immediately available to all other components. Not only the development, but also the execution of mashups is entirely performed with the help from the browser plug-in at the client side; on the server side there are only the annotations for data extraction and the stored mashup definitions.

There are also other mashup tool that we didn't cite in this list for your interest take a look at: JackBe Presto [6], Google App Engine [7] and IBM's Lotus Mashups [8].

[5] http://mashmaker.intel.com/web"
[6] "http://www.jackbe.com/"
[7] "http://code.google.com/intl/it-IT/appengine/"
[8] "http://www-01.ibm.com/software/lotus/products/mashups/"

3

The MarcoFlow Project: Overview

Nowadays, users demand for always more complex development tasks inside the browser and very sophisticated user interfaces. In the state of art we revised different approaches to service composition, showing how they lack support for user interfaces, as well as, approaches to web mashups, that lack in engineered development. The goal of MarcoFlow project is to manage the integration of UI components in the service-oriented architecture.
In this chapter, we present an overview of MarcoFlow project, taking into account all pieces of the system. Specifically, we describe the scenarios that we had taken in consideration to build our system and the motivations that had influenced our architectural choices. It is useful to remark that the contribution of this thesis is focus on the BPEL-UI.

3.1 About the project

The MarcoFlow project solves the problem of UI and service integration in the context of the service-oriented architecture for applications whose user interface is distributed over multiple web browsers. Our solution presents an innovative and integrated approach to compose and distribute web services and UI components. Moreover, it proposes a runtime architecture that supports a variety of configurations to execute applications in a distributed fashion and supporting multiple users.
Specifically, the MarcoFlow system offers a method to quickly define application in which UI components and web services are integrated that are distributed over one or more web browsers. Moreover, this project provides a system to execute these applications and to buffer the events in a distributed runtime environment in order to allow the reconstruction of synchronization/orchestration steps.

3.2 Integration in MarcoFlow

We start describing the present integration context to move then to the description of the integration context we envision in our project. The two

different scenarios are described, respectively, in fig. 3.1 and fig. 3.2. Looking at fig. 3.1, we have three different layers: the data layer, the application layer and the UI layer, which are the classical abstraction layers used in software development. Depending on its nature, each component is placed in a different layer: we can have, for instance, an RSS feed in the data layer, a web service in the application layer and a UI component in the UI layer. The integration between components of the same type is typically performed at their level of abstraction. For instance, at the data layer we use schema integration of multiple (heterogeneous) data source, at the application level orchestration among (web) services, and at the UI layer synchronization among UI components. UI components, in this context, can be seen as small stand-alone applications that can be integrated into a composite application as part of the composite layout (e.g., a Google Maps component) and that support their synchronization with other components. Differently from the previous scenario, the challenge we want to address in our project is to handle the integration of web services and UI components at the application layer. The integration scenario we envision is therefore the one depicted in fig. 3.2, where both web services and UI components are composed at the application layer; we do not explicitly focus on data components, such as RSS feeds or XML resources. For the composition of web services and UI components we want to use standard BPEL and related technologies, properly extended in order to handle also UI components. The challenge therefore becomes how to combine in the application layer both orchestration techniques (proper of web services) and synchronization techniques (proper of UI components).

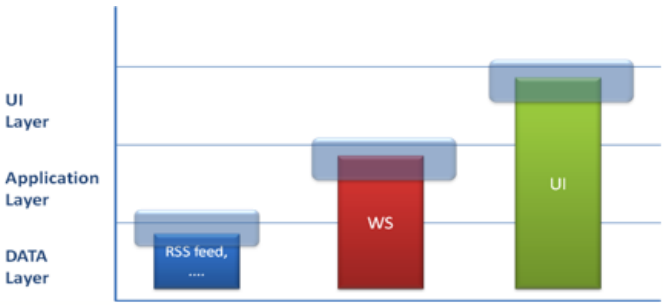

Figure 3.1: Present integration contest.

3.3 Component types

Starting from the previous contextualization of the project, in this section we detail the different types of components we can have in our framework. First of all, we define UI components in our environment. UI component are standalone applications that must be composed in a proper way to construct

Figure 3.2: Integration in MarcoFlow project.

an application UI; each UI component has an internal state and generates events to notify state changes and has operations to enact state changes. This will allow us to better understand the different types of interactions that we can have among components and to define a list of requirements. Components are described as they are seen from the outside, as black boxes. Depending on the complexity of the "business logic" that characterizes a component and its API (application programming interface), we distinguish 4 types of components (see fig. 3.3):

1. Pure service;
2. Hybrid service;
3. UI service;
4. Pure UI.

Figure 3.3: Component types.

Pure UI components are UI components without API that can be used to programmatically interact with them. On the Web, they are typically built using HTML and Javascript on the client side and JSP, PHP, and similar on the server side. That is, pure UI components are like traditional web application that we wrap integrate at the UI layer only; they cannot be synchronized with other components in a same composite application. UI service components come with a simple API that can be used to interact with them and, e.g., to synchronize them with other components. This means, UI service components can receive some data as input (e.g., a list of names) and then visualize that data in a UI (e.g., by means of a table). Yet,

they do not really process and transform the data in input before rendering them. An example of UI service component is a simple visualization widget, such as a table widget. Next, hybrid service components are UI components that come with a "complex" API, which is able to process and transform the data in input before rendering. Also, the API might be used only for processing purposes, without requiring the rendering of the data; this corresponds to the use of the component like a traditional service. Finally, pure service components correspond to classical (web) services without any UI. They can be invoked and orchestrated with other services using standard orchestration technologies. In our framework, we are especially interested in the integration of the last three types of components, which behave/act like services and, therefore, allow their orchestration/synchronization. Pure UI components can of course be used in the development of a composite application (a UI integration), though as they do not have any API that can be used for the interaction with them at the application layer, their integration can only occur at the UI layer. That is, they can be integrated into a composite layout together with all the other types of components, but they won't be able to communicate with the other components.

3.4 Scenarios

In this section, we will present the conceptual model with the help of several application scenarios, starting from very simple ones to address, to more challenging ones, that requires the integration of UI components and web services in distributed environments. The analysis of such scenarios will bring us to the definition of requirements, that have driven the project, and to design the proper models and languages in order to address these scenarios. The main idea is that of extending WSDL and BPEL and therefore having two extensions of these standards, respectively, WSDL-UI and BPEL.UI, such that they can handle not only web services, but also UI components. It is important to note that, in MarcoFlow project, UI components are exposed like WS with their descriptor, called WSDL-UI that, as we said before, is an extension of the standard WSDL. The WSDL-UI has some additional information in order to handle also UI components, like where the source code of the graphical style is located or how components react to events initiated by users. There are four scenarios that we taking in account to explain the leverages of our works, they are:

1. Orchestration of Web services only;

2. Composition of UI components only (running on the same machine);

3. Composition of UI components (running on the same machine) with web services;

4. Composition of UI components (distributed on different machines) with web services

3.4.1 Scenario 1 - Orchestration of Web services only

This scenario takes into account only the orchestration of WS. This is the classical scenario where involved web services are described by a WSDL interface and the orchestration of that is described by a BPEL process executed by a BPEL engine. In this case the BPEL-UI is a simple BPEL process, in fact no UI component is involved in this scenario. We point out that we described this scenario only to stress the fact that our framework will be able to handle also "traditional" web service composition.

3.4.2 Scenario 2 - Composition of UI components only (running on the same machine)

As we said before, UI components are defined like Web services having a set of invoke, reply and receive operations, and, in the same way, they are composed at design time. Each UI component exposes a WSDL-UI to describe its functionalities and the way to interact with it. This scenario takes into account only the composition of UI components. By way of example, suppose we want to create a web application where we mash different UI components, the aim of such an application is showing information about national park in USA. We have three UI components to mash up: (1) one which allows looking for parks in the USA by name, location, activity and topic, (2) Google Maps which will allows to show the park location, (3) a Flickr-based one that will shows pictures of the selected park. The architecture of the scenario presented is depicted in fig. 3.4. At design time, the composer, using the WSDL-UI describing the UI components, creates a composition. Then, the BPEL-UI process defines the interaction between UI-components. As we said before, a BPEL-UI is a standard BPEL process with some additional information that allows at runtime the loading of UI components. The BPEL-UI process cannot be given in input to a BPEL engine as it is, so a middle component - the BPEL compiler - has to be introduced. The compiler takes in input the BPEL-UI, processes it and gives in output two files: the BPEL process and the UIC configuration. The first one is a standard BPEL process that will be processed by the BPEL engine; the latter is a configuration file that contains all information related to UI components. The UIC container receives as input the UIC configuration and retrieves from it all the information needed for the instantiation of components and graphical rendering of that. Moreover, the UIC container includes a web server needed for loading the UI Application. Every time a user invokes a UI application (box App in the figure), each component is loaded through an interaction with the UIC container. When the user gen-

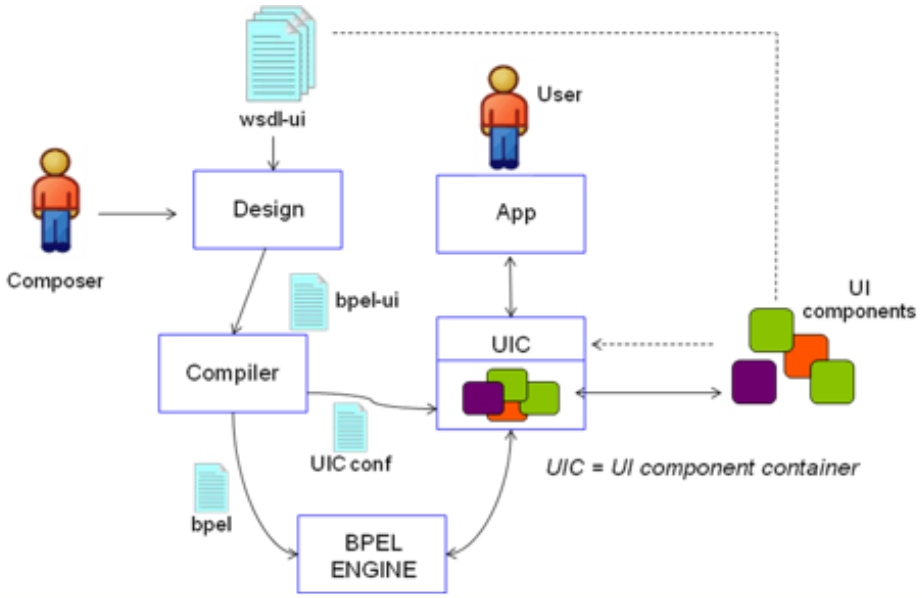

Figure 3.4: Composition of UI components only (running on the same machine).

erates an event, the UIC container communicates with the BPEL engine to execute the associated BPEL process, like the displaying of a picture in the previous example. UI components are available on the Web, but in order to run them - unlike regular web services - you need to load them into you UIC, i.e., you need to instantiate them in your UIC. We notice that the UIC can be seen as a regular web server running the instances of the UI components that also have internally some own logic that allows to put together the UI components as described in the UIC configuration. Then, the actual UI is rendered in the client browser as for any standard web application. We also point out that we are considering services that can have both a UI and an API side.

3.4.3 Scenario 3 - Composition of UI components (running on the same machine) with web services

In this scenario we analyse the composition of web services and UI-components that are in a single web page (running on the same machine). The architecture of this scenario is depicted in fig. 3.5. The architecture is very similar to the one presented before except for the fact that the designer composes his BPEL-UI process using both WSDL-UI (for UI components) and traditional WSDL (for web services). As said before, these descriptors are different because of some annotations that are useful to distinguish UI components from WS. At design time it makes no difference, in fact these files

have the same structure, so the designer does not differentiate if a WSDL is related to a WS or a UI component. At runtime, instead, one cannot disregard the difference between these two; for this reason, the WSDL and BPEL process need to be extended with some annotations. As we see in fig. 3.5, if the WSDL is related to a WS, the BPEL engine can interact directly with it; otherwise the UIC container has to be involved in the process. This means that the BPEL-UI is sent to the compiler, which is then linked to the BPEL engine and the UIC container, the first one take care of the web service composition, while the second one take care of the UI components.

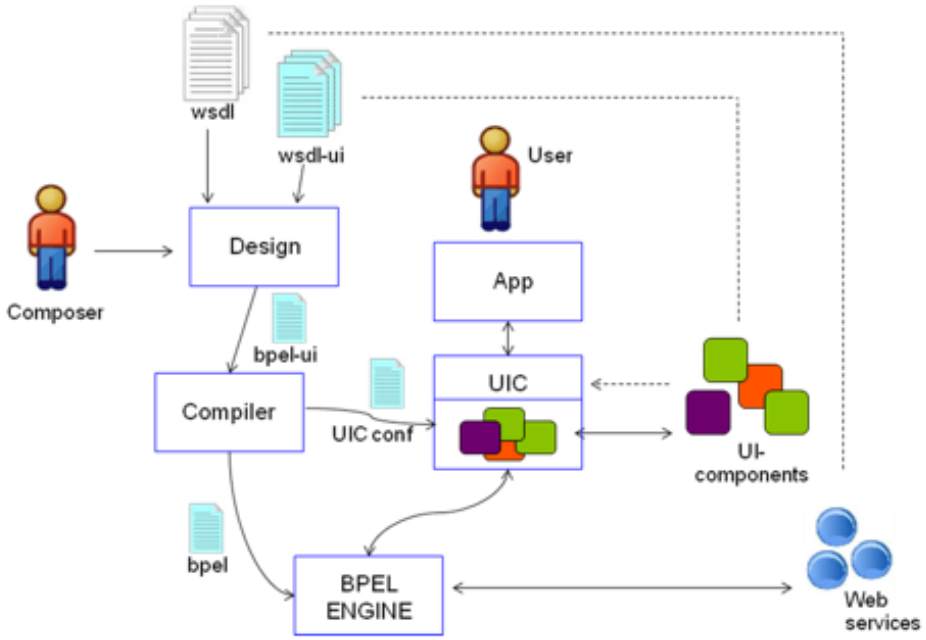

Figure 3.5: Composition of UI components (running on the same machine) with web services).

3.4.4 Scenario 4 - Composition of UI components (distributed on different machines) with web services

In this scenario, we analyse the composition of several UI components, each group distributed on different web pages, and various web services. In this case could be useful discuss the composition logic of this scenario, depicted in fig. 3.6, where we have UI components running on two different machines that interact with several web services (S1... S5 in the figure). Then, the BPEL-UI process we define is needed in order to orchestrate them. The scenario could be the one in which different UI components are on machine 1, then one of these UI components can invoke an external service S1, which can invoke in turn an other service S2. The service S2 can then generate

some events for a UI component running on a different machine (UI5 on machine 2) and then other UI components on the same machine can call different services. In our vision, the entire process is orchestrated thanks to the BPEL-UI process we are able to define. It is interesting to notice

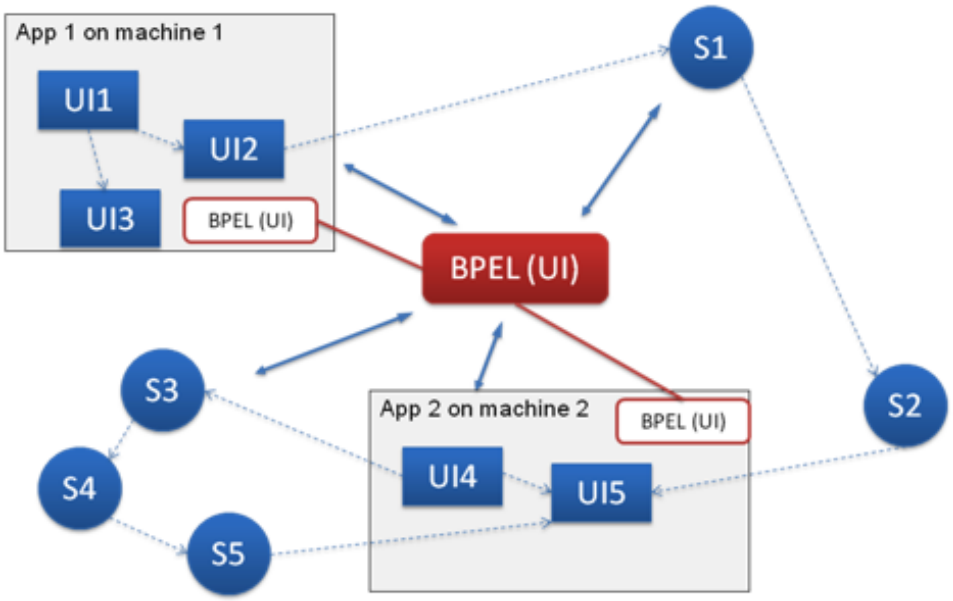

Figure 3.6: Composition logic of UI components (distributed on different machines) with web services.

that pages can be loaded at different time, so a process can stop its activity waiting for a new user's event to happen. There can be two different sub-cases of this scenario: one in which the BPEL engine is centralized and the other in which there are two or more communicating BPEL engines. The architecture of the former use case is depicted in fig. 3.7, while the architecture of the latter is depicted in fig. 3.8. The architecture in fig. 3.7 presents many UIC containers, one for each web page, and only one centralized BPEL engine; the composition logic is similar to the one presented before, excepted for the fact that now we have several UIC containers, as they have to load different UI components running on different machines. Obviously, when a UI-component should be load, the engine has to send the request to the proper UIC container. More complex is the second case, in which there are several BPEL processes (and so BPEL engines) that interact with each other. The architecture is depicted in fig. 3.8. Now, after presenting these different scenarios, we can address the problem of the protocol stack and how this should be rearranged to integrate also the management of UI-components; in fig. 3.9 is depicted the actual protocol stack, while in fig. 3.10 is depicted the protocol stack we envision in order to realize our framework, which will be able to handle both web services and UI com-

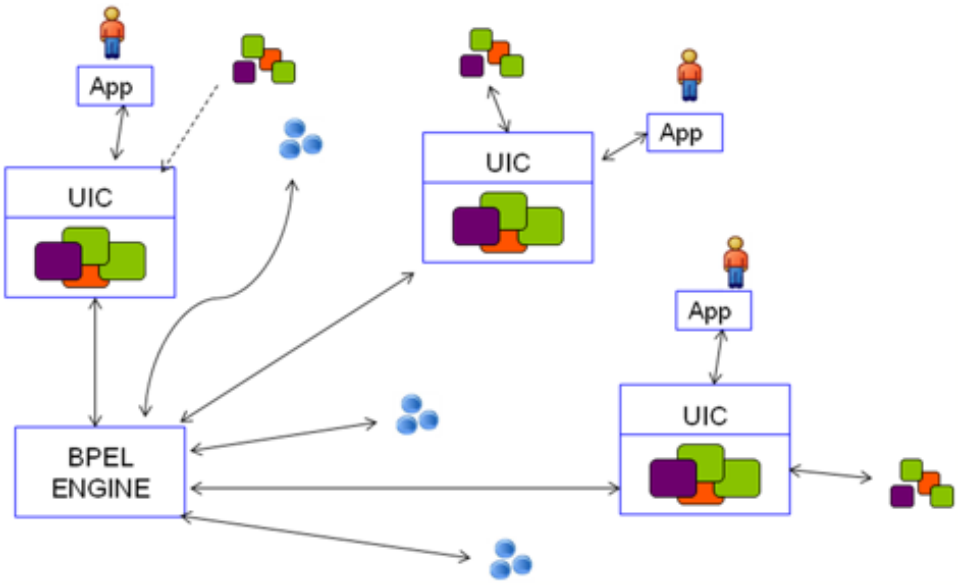

Figure 3.7: Composition logic of UI components (distributed on different machines) with web services and a centralized BPEL engine.

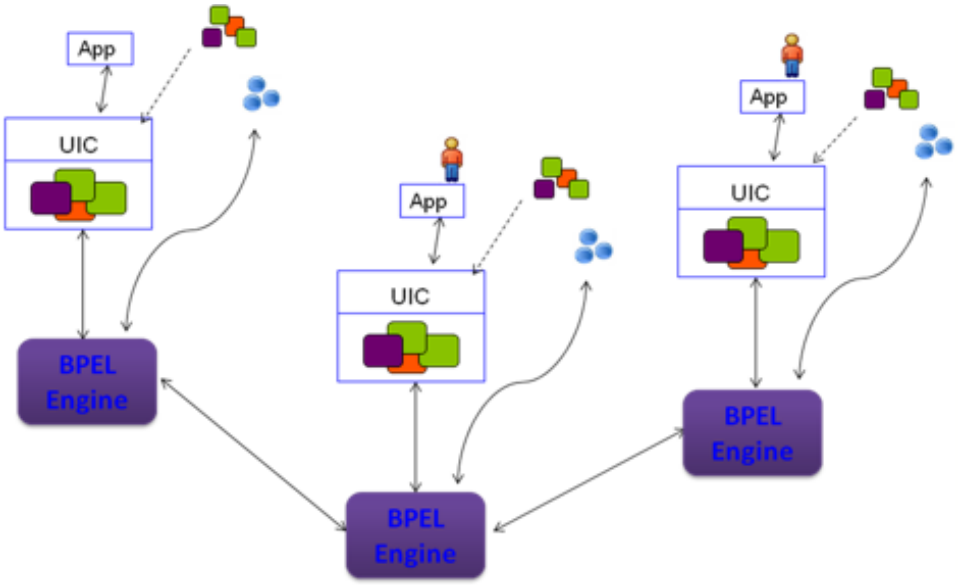

Figure 3.8: Composition logic of UI components (distributed on different machines) with web services and and multiple, distributed BPEL engines.

ponents. As shown in fig. 3.10, web services are described, as usual, by a WSDL descriptor, while UI components are described by a WSDL-UI descriptor. Both descriptors are used to create the BPEL-UI process, where

the orchestration is defined, then the BPEL-UI is split in two specifications, a traditional BPEL process and a UIC configuration, each one is handle, respectively, by a BPEL engine (able to execute deployed business BPEL process) and a UI component (able to instantiate components and graphical rendering them in UIC container). The following sections have the purpose

Figure 3.9: The actual protocol stack.

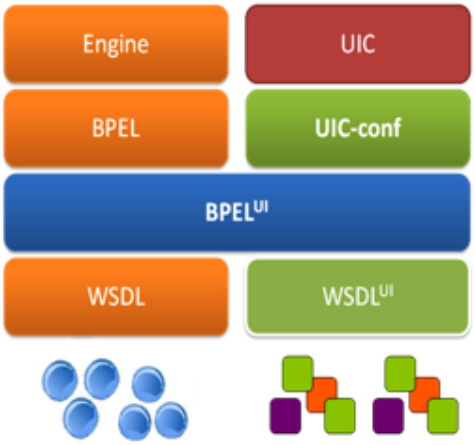

Figure 3.10: Protocol stack to handle web services and UI components.

to give an overview of MarcoFlow project. Specifically, we analize step-by-step the facts and the reasons that taked us to develop/envelop this system. All parts of MarcoFlow architecture are taken into account: BPEL-UI, UI engine server and UI engine client.

3.5 BPEL-UI

Generally, BPEL contains constructs that are sufficient for expressing business processes. In this case, however, it could be necessary to extend the BPEL language with additional constructs. A deeper analysis can help us to understand which are the constructs that must be estended and which concepts are needed to introduce. Specifically, we start from very simple scenario to address to more challenging ones with the aim to phase in the innovation that each scenario suggests.

Scenario 1. This scenario takes into account only the orchestration of WS that is described by a BPEL process executed by a BPEL engine.

Scenario 2. This scenario takes into account only the composition of UI components and, as everyone knows, this composition cannot be described by a BPEL process. It may be useful to remember that in MarcoFlow

project the UI components are exposed like WSs by their WSDL-UIs. This means that the synchronization of UI components and the interaction of web services can be handled by the same partner link construct. For this reason, we must only extend the syntax and semantic value of the partner link element, without any semantic alteration. We always keep in mind that the first of them implies the introduction of new information. In this context, it is needed to know if a partner link is referred to a UI component and, in this case, the list of configuration properties (optional). This last information has the use to include the component setup information such as layout and/or configuration parameters. Moreover, as we said above, MarcoFlow platform gives us the opportunity to create the UI for a certain application. This innovation aspect may suppose the existence of a new construct, called page, with the following new features:

- a name as unique identifier;
- a pathname of an HTML template. This information is useful to retrieve the list of the logical containers (HTML div) in which a UI component can be shown;
- the list of the logical containers;

At this point, we should only define the relation between partner link and page construct. Interestingly, the UI component referred to a partner link should be visualized into a page, so it is sufficiently to introduce new features into the partner link construct. Specifically, the name of the page and the name of the logical container in which this UI component must be shown. In this manner, a developer can create very sophisticated web pages in a few easy steps.

Scenario 3. In this scenario we analyse the composition of web services and UI components that are in a single web page (running on the same machine). No more concepts are needed to handle this more complex scenario.

Scenario 4. In this scenario, we analyse the composition of several UI-components, each group distributed on different web pages, and various web services. Interestingly, there can be two different sub-cases of this scenario: one in which the BPEL engine is centralized and the other in which there are two or more communicating BPEL engines. In consideration of this fact, the partner link construct should be contains another one attribute to identify the BPEL engine linked to a certain UI component.

Now, we have defined a high level language to handle the integration of UI components in the service-oriented architecture. In addiction, MarcoFlow system supports the development phase providing an extended BPEL editor

to compose BPEL-UI processes

This piece of MarcoFlow project is the contribution of Valeria Vianne's thesis.

3.6 UI Engine Server

This paraghraph is a fast summary of the whole thesis work.

In this section we want to explain the considerations that lead us to find and implement the solution for the UI engine. As said, we would take the technologies solution existents as they are, without invasive modification. In example, we wont to rewrite the code of the BPEL engine to have coexistence of components and services possible, we will find out a solution that can be used with all the process engines, which is also adaptive with the UI framework and is not expensive in the whole architecture.

The scenarios proposed are good method to explain, step by step, which things we have to solve and what the solution needs. The *first scenario*, orchestration of web services only, does not provide new claims to satisfy, the solutions available already satisfied this scenario. The *second* and also the *third* scenario, instead, introduce the UI components inside the orchestrator language. In the discussion of these two scenarios we introduced a black box called UIC, in this box we would have UI components and services, or something similar, for the coordination of the data between the two side of the communication. In the composition of the scenario we have also explained that the UI components, modeled with the WSDL-UI files, are trade by the editor like standard web services and the process interact with them in the same way. The UI engine client side solution is provided by a framework that has a full integration of web services in an AJAX technology, it especially support, as well, the RESTful services. Substantially we have to find out a solution that is able to handle the communications among UI components and BPEL processes. The components are managed in the processes like standard services; these component, by client framework, are also able to manage services. These considerations led us a possible solution based on services that is able to handle the communications; in the both sides of the conversation we have a support of this possible solution.

The solution based on services is done also in consideration of the last scenario, the distributed one. In this case, the uses of services give the possibility to have this solution almost for free. The services are the base for all the distributed application solutions; especially in BPEL language based scenario they perform a decisive role in the evolution of the process data.

The potentiality of this language, in the coordination of services, and in this case also components among services, permits to have the last scenario without effort of the developer. The orchestration is modeled in the process with the services files (WSDL file o WSDL-UI file), the coordination of the messages, instead is managed by our solution that have to handle the message from both side.

The problems involved in a solution based on services are raised by the technologies used and the logical implication of a distributed application and stateless objects, like the services are. The challenges of this solution is basied on the coordination of the messages: we have to provide a solution that brings the messages from the right UI components user associated to the right process and vice versa. This is not a trivial problem to solve, there are many cases and different solution, but we want the once that is flexible enough to be maintained and construct by an automatic generator of services. Our solution has to be also able to manage the different instances of application in the whole context, the compositions are web based on the client side, so they can be executed by more actors at the same time. This scenario implies a multiple instance of the same application and also multiple messages to handle and to right deliver. The coordination inter page (coordination among different users browsers) is not a trivial problem to find out and to provide, also here there is a coordination problem of messages. The coordination is not the only one problem that we encountered during the research of the solution, also the technological solution available since now involves other challenges to solve. The components for the final user are created to be maintained by a standalone application and they do not usually provide a method to be managed from a distributed context. Besides, they do not support (in our case is the JavaScript) the possibility to receive messages immediately from a web service, and this is a problem. There can be many messages created from the process for a UI component of an application, and there is no way to makes them available and received by the component in a real time fashion with standard and robust solution.

To solve all these requirements and challenges, we provide the solution based on services. The services involved in the final version of the UI engine are differentiated in base at their goal and their implementation. In the communication paradigm we have only two ways that a message can follow and they are: BPELtoUI communication, from process to component and UItoBPEL communication from component to the process. In both of cases theres the coordination and correlation problem to solve, in the first one case the problem is solved by our UI engine. We provide storage of the events both to avoid the technology gap of the UI components that cannot receive immediately the message, and to create a method to a right delivering of the data. In our storage (is a buffer of event) we insert also a key, which

identifies the instance of the application; each time an event is inserted or is extracted from the buffer the key is used to select the right event for the right component. Also the component is discriminated by values inside the event and buffer, namely we are able to distinguish also the different components inside a page. On the other side, from the UI component to the BPEL process, the correlation of messages is provided by the orchestration language and the orchestration engine, in this case the BPEL engine already contains a method (correlation) that is used for this propose. Indeed our solution for the UI engine is inspired by this mechanism, and we readapted this idea inside ours one.

Finally, to achieve the goal of distributed application, we provide a method to deploy the engine that permits to generate component for different machines in different places. We create a compiler with a configuration that creates the services and deploy them on different machine. This allows the creation of the environment for the last scenario.

3.7 UI Engine Client

To develop the client-side runtime environment of the MarcoFlow platform, we need to approach some issues that are peculiar of our UI orchestration approach:

- manage the data flow among all the components;
- manage the communication between the UI components and the BPEL process;
- carry out the communication between UI components;
- rendering of UI components.

In order to develop a framework that accomplishes our requirements we investigated the state of art about integration at the presentation layer. Hundreds of examples of presentation integrations exist today in the form of web mashups. Web mashups perform integrations both at the application level and at the presentation level. This is more than we need because our goal for the client-side runtime environment is obtaining only presentation level integration, while we leave application level integration to the BPEL engine. In [20] the authors introduced a first, end-user-oriented framework for integration at the presentation level, that is, integration of components by combining their presentation frontends rather than their application logic or data. This is similar to what we need in the MarcoFlow runtime environment. In [20] and in its extension in [4], the authors proposed a whole

framework inspired by lessons learned from application integration, appropriately modified to account for the specificity of the UI integration problem. The framework includes also a tool that allows one to drag and drop components on a canvas and quickly specify the UI integration logic so that a complex application can be built by aggregating components with minimal development effort. This kind of design time support is out of our interest because we focus on BPEL-centric composition of UI components and web services and we target skilled developers. Yet, the approach in [4] also provides an abstract component model to specify characteristics and behaviors of presentation components and propose an event-based composition model to specify the composition logic. Components and composition are described by means of a simple XML-based language. We adapt the lessons learned in this approach to the client-side runtime environment in MarcoFlow.

In order to develop this client-side environment we take in account several properties to obtain. A first consideration regard the runtime location that in our case is client-side for the execution of the composition that involved only UI components. Instead if the composition includes also communications with the BPEL process the client-side environment is not enough and a server-side runtime its necessary. Obviously the two runtime have to interact and how we see later we developed two special UI components to obtain this communication flow. The second property is about the system requirements and the possibility to force the end-user to adopt some additional browser plug-ins or extensions. All the components, also the ones required to communicate with BPEL process are implemented in JavaScript and this does not pose any particular system requirements on the client. The last aspect to consider is about the scalability. It can be considered from two perspectives: in the number of models (compositions), or in the number of users. If the composition include only UI components we need no special attention because the composition is executed on the client and therefore there is no bottleneck. On the other case if a server-side runtime its necessary we could soffer of some scalability problem related to the number of instances and, hence, to the number of users. In this latter case the MarcoFlow system doesn't care about scalability because adopt an engine-based runtime , that include itself techniques to obtain scalability.

The client-side runtime environment we propose for MarcoFlow is an adaptation and extension of the framework proposed in [4]. In particular we adapted the abstract component model and the composition model to the new requirements emerging from the UI orchestration scenarios. Indeed we eliminated the abstract component descriptor, formulated in the mashArt Descriptor Language(MDL) and substituted it with an extended WSDL file. Furthermore we modified the XML-based composition language, the Universal Composition Language (UCL)[4], which operates on MDL de-

scriptors only. Moreover the framework itself was modified to permit an indirect communication to the BPEL process.As we said before, in order to allow this interaction with the BPEL process we introduced two special components that we called Eventforwarder and NotificationHandler. These two components are, like the others UI components, developed in JavaScript but differently from the other ones the designer can't use them in the composition and they don't interact with the final user. They are system components and so they don't have a rendering aspect.

This piece of MarcoFlow project is the contribution of Michele Iannotta's thesis.

4

The MarcoFlow framework

4.1 Introduction

This section introduces the work of thesis and especially the framework's communications on the server side part. To have a better understanding the figure 4.1 shows the whole architecture of MarcoFlow project, the blue objects highlight the artifacts thought and developed during this thesis (not all are discussed here), in the light blue the other artifacts created by our system or by the developer and in white the other file or objects that already exists. The solution proposed tries to achieve the final goal of the project that is an orchestration of UI components. Mainly, the problems encountered are technical and logical. In this chapter, we trade and explain how a UI component can be used in an orchestration language, and how an orchestrator process can communicates with the UI components and the user associated. We will take a look through the solution, in theory and in practice, how the different operations present into the architecture are managed by our server side framework. We will show how we transform the paradigm of an orchestration language for service into an orchestration language for services and UI components; a language which is able to manage event from users and coordinates user interfaces. We also provide an explanation about the idea behind the solution, how we manage and coordinate the data that pass among the orchestration process and interfaces, how the UI components can be saw as services and used inside a process that manages services.

4.2 The idea behind the solution

We would start from the idea that we have in mind to satisfy this problem and achieve the final goal. The orchestration languages, is commonly used to manage web services, in the last years some extensions are introduced to avoid the lacks of the BPEL language. Some of these extensions are done to models the human-interaction; a workflow has an intrinsically presence of human tasks. All these extensions, anyway, don't take care about the final application; they don't permit to describe the final user interface. This is

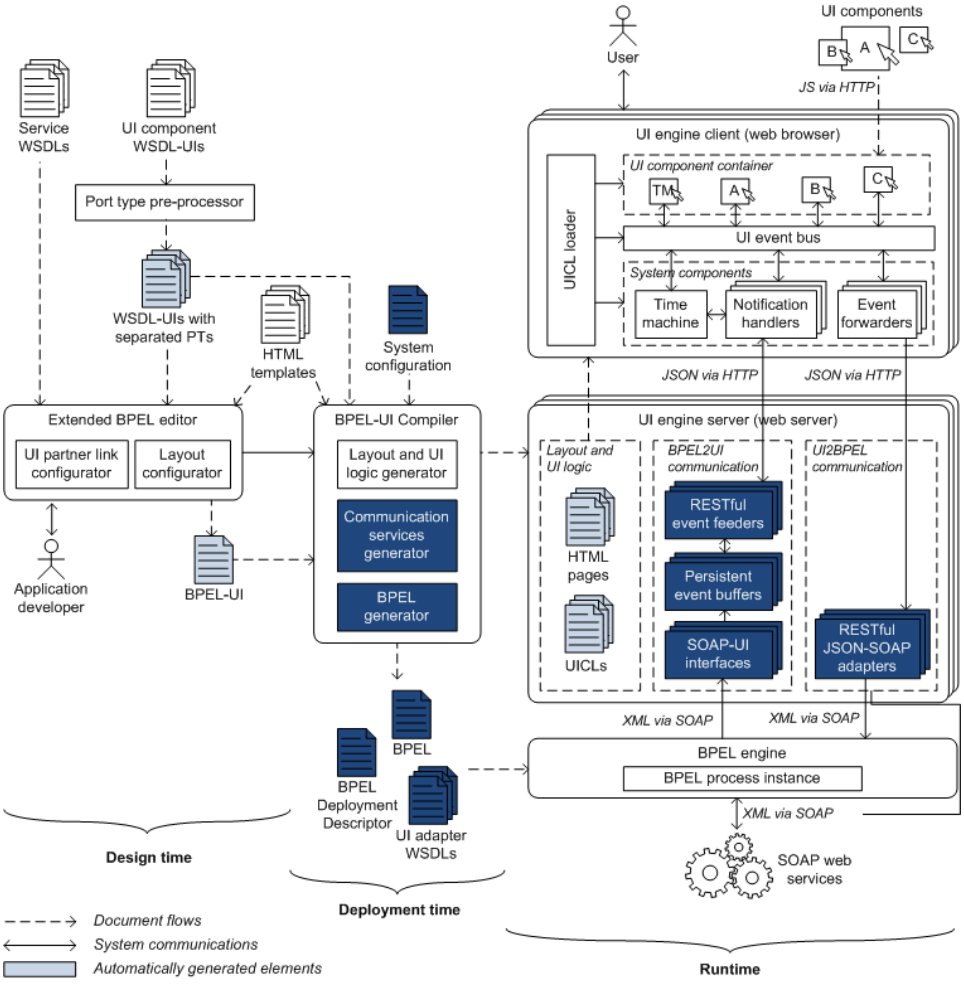

Figure 4.1: Complete architecture of MarcoFlow project

one of our final goal.
The workflow orchestration language that we choose, the BPEL language, was thought to orchestrate web services, and it does this as well. We want to extend this language, not modify it; it has to be an orchestrator for components and services. We also want to use all the technologies at their standard as much as possible, so we won't to create a special engine for the language that is able to manage our new definition of process, but we want to keep the existing technologies and create over there a new layer that is able to handle the difference between the orchestration languages and the final user application.

Saw the peculiarities of this language, and saw ours ideas, we decided to build a server side layer that is a manger for communications between

process and components. The language chosen as orchestrator of components was created to manage services. Our idea, in fact, is to create the UI components describing them like a web service (WSDL file plus some extra internal extensions), in this way they can be used like standard web services. Following this idea we have a process that interacts with services, this is nothing new, the news is in the fact that some services are not a classical web service but are wrapper for UI components. On the other side, in the client side, we have components inside a web browser managed in the process like services. This is the idea of the solution; the main problem is how we can provide a method, which permits the communication, from process to components, through service.

The base of the idea is to use services on the both parts of the communication, the BPEL language manages services, it orchestrates web services, it was made for achieve this goal. The UI components, instead, usually don't interact with web services directly from the client layer, or better, in the past they didn't. With the advent of the widely uses of internet, some new lightness technologies was introduced, the principal are the AJAX technology and the RESTFul services. These two, both together, are commonly used to construct rich and dynamic application directly on the client side, without extra additional layer. From these considerations we started to think about a layer that is able to make in communications the components and process. On a side of this framework, we put classical services (SOAP services) that can communicate with the BPEL process; on the other side, instead, we put the services for the client side framework (RESTFul). Our client side framework is made by AJAX and JavaScript, and it supports as well the RESTFul communication. This is our solution: create a series of services that communicate that are the middle layer between process and components. This solution, anyway, is not trivial as it appears, there are some underground lacks that we have to avoid. For giving an idea: there's the correlation of messages to maintain, and the problem about how to send messages to a UI component. In this chapter we will try to explain this idea at best and the solution for all the problems encountered.

4.3 Technology background

Before start to talk deeply about the proposed solution, we want to clarify some aspects about the operations of our system.

The BPEL languages, in according to the WSDL definition 1.1 [3], manages four different message patterns, in WSDL 2.0 there was introduced also other patterns for messages but we treat only the classical four, they are:

- *One-way* This operation is composed by input parameter only, there's no output over this function. Typically is used to send messages from

client to the service.

- *Notification* This is the mirroring of the one-way operation, has only the output parameter, typically is used by the service to send messages at the client.

- *Request-response* This is composed by input and output parameters, in this order. The operation is used to send a message to the services and receive the response; this set of operations is done synchrony. In other words, the connection still remains open until the response, in a BPEL environment this means that the process stay blocked until the response.

- *Solicit-response* Like the Notification is the opposite of the One-way, also the Solicit-Response is the opposite of Request-Response. For the "opposite" we means opposite in the action done by the actors involved. In this case, the request is done by the service (and it's named solicit) and the response is done by the client.

These are the standard message patterns used in a client-service communication. In our project we introduced the WSDL language as descriptor for UI components, the component is different by definition respect service. Usually these UIs have only operations and events. To permit these new uses of WSDL document, we allowed only two patterns instead four, we allow only: the *One-way* and *Notification*. This limitation regards merely the WSDL-UI file, the WSDL description of services can sill model all the patterns available, and the BPEL language is able to manage them like it has ever done.

4.4 Architecture of Service Side

Firstly, we have to keep in mind what claims our architecture has to satisfy. The whole goal of this project is to orchestrate UI components and services. The orchestration of services is a goal already done by a lot of technologies, the most important and used is probably the BPEL languages, and we chose this language as orchestrator. With this choice, and saw the capability of this language to manage services, we decide to use services for describe components, in this way, they can be simply managed by the process. The main problem to avoid is due to technologies used in these two kinds of environment that are services and components. These two worlds, services and UI components, are not logically similar, the first one usually offers information, the second one are used to keep data from user, they are typically a front end for the user and they don't contain computational logics.

The technologies presented nowadays for services are not directly compatible with these available for the components; the exchanges of data among services and components typically need some extra effort by the user, or developer. In our solution we decided to use the most common technologies that are available in this moment, the chosen of BPEL languages implies that the services used have a WSDL descriptor and talk via SOAP messages. For the components part, with the large growth in the last years, we decide to use JavaScript components and AJAX technologies for the communications. These choices, however, don't satisfy the final goal, the communication between processes and components is not providing directly; with this architecture we try to avoid this gap. Anyhow, this is not the only one problem that we have; there are also other limitations that we want to avoid. The UI components, since now, don't offer a method to receive immediately messages, the data inside a component can be inserted by the user or can be taken by the component itself with a scheduled operation, the second one is commonly called *polling*. The other request that are always intrinsically in a SOA system are also taken in account, the uses of services as solution for managing components, on different browsers, permits to have a distributed environment and this is a claim that we want to satisfy.
There's another point to take in account: the BPEL-UI language can define a long running process, using different actors that communicate through it; there will be the possibility that not all of them are online when the messages are exchanged, can be happen that a message is sent to an offline user. We need to store these messages and preserve them for a second moment, when the user becomes online.

All these considerations were taken in account when we built up the architecture. We try to use, as much as possible standards technologies, our solution for the architecture server side is shown in figure 4.2. In this figure we kept all the items involved in the communication, the other objects that are not necessary in this part are removed or hidden, in this way the reader can have a better understanding about the architectural part. How is simply to understand by the figure, there are in practice two kinds of communications that pass through the server, they are named respectively *BPEL2UI* and *UI2BPEL*, and each communications used different artifacts. The first one flow of communication is used to manage the messages from the process to the components; the second one is the mirroring (from components to process). In this way we avoid the first gap that was: how the two sides can communicate each other. The second one problem, which was the limitation of components in reception of data, is avoided by storage of events (the component is visible on the left side part of the image), in this way the components can retrieve the data performing a standard polling. All these things are explained better here below.

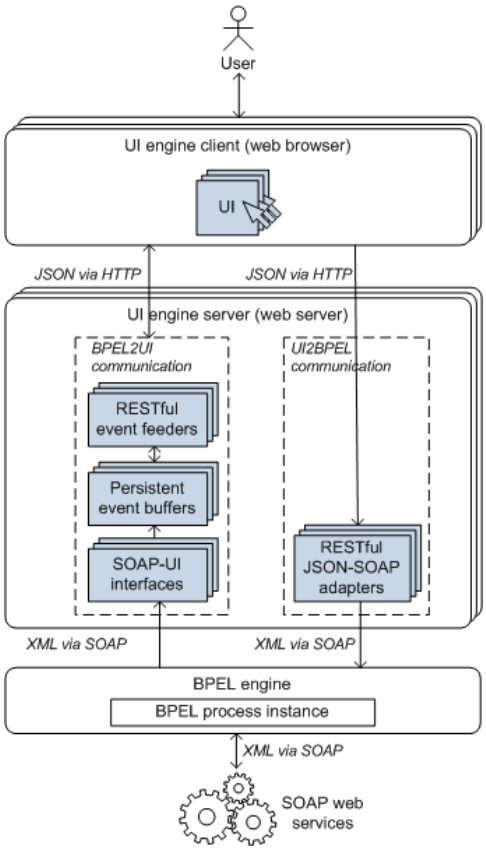

Figure 4.2: Runtime architecture

4.5 Communication paradigm

As said in the section 4.1, the only operations that a UI component can performs are: One-Way or Notification, these things imply an implicitly asyncronicity of the system. This "problem" is not given only by the operations' style, but also, as said, due to a lacks of the technologies involved. A BPEL process can obviously receives and sends messages, and this is a matter of fact. Vice versa the UI parts are also able to send messages, but it has no way to wait messages from other actors, this means that a notification cannot be really performed. There's a tricky solution that permits to execute these operations, in fact, the UI component can make a call to check, by itself, if some new message is arrived; this is common called a polling operation.

We introduced, in chapter 4.4, only two types of communications named *BPELtoUI* and *UItoBPEL*, this because only these two send the messages through the server. However, there's also another communication that is developed in the BPEL-UI process but doesn't needs a server to be executed:

this is the UI-UI communication (from a component to another one component of the same page). This communication is related with the works of this thesis but is done and explained in another one thesis. Anyway, we want to spend a little bit of time over this communicational pattern; this can be useful to understand the other communications and to avoid some lacks when the example will be explained.

The UI-UI communication is written inside the BPEL-UI process, with the standard components of a classical BPEL process, it's identifies by a specific pattern and specific rules of composition. In this moment we consider a communication from/to a user-interface, which is a UI-UI communication, when: (i) it respects the *receive-assign-invoke* flow operations and (ii) the services involved in this communication are UI components. This diversification permits to find this kind of communications and move them from the BPEL engine directly into the client side framework, in this manner these operations are performed inside the browser and no data are sent or received by the server. This peculiarity removes effort from the server, and save some net's traffic. This separation is obviously useful, but is complext to perform, especially in the discovering of the right pieces to remove. There are a lot of cases where the communication cannot be extrapolated and placed in the client side. The composition language is very powerful, and in this case is not a good thing, it allows using different styles of definitions about the manipulation of data, a list might be: simply manipulation, access by with element-base, access by a message base, XPath definition, CDATA and so on [15]. There is also the problem that a variable can be reused in another flow of the process, or it values can be used for other assigns and so on. In all these cases these operations have to remains on the process, otherwise we will have a loss of data in the process that probably implies a non working process.

The other two types of communications, which are: BPEL to UI communication and UI to BPEL communication will be explain in the next section.

4.5.1 UI's operations and BPEL's operations

There's a thing that we want to explain before proceed. First of all there's distinguish to do about the message patterns inside a WSDL-UI document. We added a semantic to each pattern allowed, the semantic is given by the commonly uses that is done with these words in a UI components environment: the one-way is called *operation* (operation with only input for the component, so messages goes from BPEL process to UI component) and the notification are called *event* (these are operation with only output for the component, so their messages are generated by the component and sent to

the process). This diversification is given from the component point of view: the events are something generated inside the component and forwarded to other actor, vice versa operation is something that can be received by the component.

Anyway our vision of UI component is closed to the vision of a web service; the process can receive data from UI component (an event) or can contact the component in order to send it some data (operation). However, the UI element is only a frontend for the user, it contains only logics for the management of the results and the way about to present them. A component is usually not able to do computation by itself, it shows data and permits to insert data that will be send at the controller. We have to think the components in this way: elements that receive operations from the process and that are able to generate events and sends events to the process. From a practical point of view, the BPEL process has to expose all the *events* of each UI component involved into the process, in this way it can receives messages. The operation, instead, are called by the process and read by the component, in this case the provider of the operation listener is our service layer.

4.5.2 UI to BPEL communication

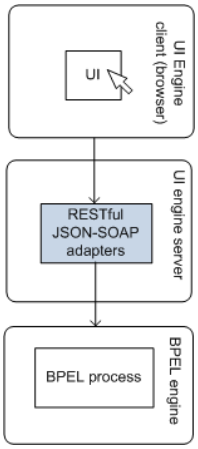

Figure 4.3: UI to BPEL communication

Figure 4.3 shows the artifacts needed for the UI to BPEL communication. The goal of this communication is to be able to receives the *events* generates from the components on the client's browser and reply them to the process, in a correct way. The communication from components to process is simple as concept, as already known the BPEL process exposes a service that receives messages from each UI element. In practice, there's a web

service, and this web service is used as receiver for messages, in our case, these messages are sent by the UI component. Typically, when someone wants to communicate to a web service, it needs a *stub* of the service or a system that creates a SOAP message and sends it to the right endpoint. We adopt a modification of the second solution; when we compile the BPEL-UI process we create the service, this service takes data from the process and it's able to construct a caller for some operation of this process. To be a little more precise, the service takes all the events that are described in the WSDL-UI document and creates a structure that can: (i) handle each event messages arrived, (ii) build up a message for the process, (iii) and send the message to the correct endpoint. Obviously, not all the data are already available at creation time in order to create a message that can be send and understanding by the process, it needs also the data that has to put inside the messages, the values are given at runtime by the user through the component. In fact, when an event is generate by the UI object with the collaboration of the user, is the *Restful JSON-SOAP Adapter* the receiver of this message; the message contains the last data used to build up, at runtime, the SOAP message for the process. After this, the message is sent at the correct BPEL's endpoint (the endpoints' explanation is treat more deeply in the section 4.6). In this way the process can receive the message and it can starts or continues its execution.
Anyhow, in this communication, we don't need any extra tricky component, the flow of the communication is clear and the technologies involved allow to perform exactly what we have to do. There's no extra gap to avoid, the process waiting for messages, we don't need to store any kind of messages, the asyncronicity is managed by the process and not by our framework. The only things to do is to provide the correct SOAP message, this message has to contains all the data, especially a field for the correlation used by the BPEL engine, this field is used to manage the instance of the process inside the framework (this will explain better later).

4.5.3 BPEL to UI communication

The BPEL to UI operation is a little bit more complex than the operation explained before. The cause is simple: there's the problem of the asyncronicity. For a fast recap: The process can sends and receives data through its services, so here there's no problem; vice versa the UI element is able to send data, but the reception activity is not performed directly, so it doesn't have a method to receive immediately the data. The reception process has to be performed in a different way: the UI component has to ask at the server if some new messages are available for it. The goal of this communication's flow is to avoid this implicit asyncronicity given by the technologies. For this intent we adopt a solution that allows us to store events and provide them when the UI objects make a call. We create a receiver for messages of the

process, and these messages are stored in a buffer like events. On the other side, there will be a component that performs a pooling call each fixed time interval, this component will ask at the server if some events is available for a determinate element, if so the service will provides this message.

Persistent Event Buffer

Before starting the explanation in detail of this message pattern we will explain better the *Event Buffer* (the middle components in the 4.4) and the motivation its uses.

In section 4.1 we explained our choices about the message patterns allowed and we decided to let to use only one-way/notification operations. So we don't provide, and we'll probably never provide a request-response operation, for various motivations. First of all, the request-response is a blocking operation, in other words when a user invoke a request-response operation, the applications has to waits the reply before the execution of other operations, this implies a total block. The second reason is more practical, the UI element cannot receive directly messages from process, in fact: is the process that sends messages and then, the component, searches if some message is arrived, this is normally called a polling system. If we allow the request-response pattern we will have slowly application, the cause is the big amount of waiting time (between each sends and each pooling operations). However, the UI objects cannot execute the pooling each millisecond, so the operation is performed each slot of time (this slot is variable, and usually is in order of seconds). In this gap, which is from process that sends message and the polling of the UI component, there's a system that store the messages. This is done to preserve all the communication and to avoid loses of messages due to the asynchronicity of the system.
There's another thing to explain, and another valid motivation why we use this buffer. The one-way message is by definition asynchronous, this means that the BPEL process sends the message, and after that it continues its flows performing all the other operations. In order to show the message, sent by the process, in the component, there's a polling that has to be executed by the UIEngineClient. As said, the polling operation cannot be performed each millisecond, this means that sometimes the BPEL process sends more than one messages in this period (the period between two polls) and we, and also the user we think, don't want to lose these message. If we store only one message each time, and the messages will be two or more in a period, we will lose part of them. To avoid this lack we have decided to store all the messages in our event buffer system, in this way all the messages from the BPEL process can be read by the UI component lossless.
These are not all the motivations that justifies the persistent event buffer. Our BPEL-UI process allows defining a long-run collaboration's process,

and in an environment like this an event buffer is mandatory. I.e. if more than one actor is involved in a process, there's a possibility that not all the actors are online when the process started and data are sent. There's the possibility that some messages are sent to actor that are not online, if we don't store the messages we will lose them and the actor will never receives. In our solution this scenario works, the actor can communicates asynchrony and receive the message when he becomes online. The storage of the message gives us an opportunity to make a lot of features useful for the user, like the time machine or the restoring of previous state. We have all the messages stored in the buffer, so we can navigate through it data and then show at the user the messages in a timeline.

In the list here below there's an explanation and a representation of the *Event Buffer*. The name of the event buffer is the name of the component, so there will be a buffer for each component. Its fields are:

- **application instance key** This field is used to manage the correct application instance, as hint before, a process can be long running, so if an user goes offline and after a period he wants to restart his work, he has to use a determinate key to identify the application. We use this key to manage the different instance of an application. A more deeply explanation is present in the section 4.6.2 where we explain how the system handles the difference instances of the same application.
- **component instance name** This is the name of the instance, it's a pragmatic field. We build the event buffer in order to have once for each UI component, so each instance of a single component is contained in an event buffer. Using this field we can retrieve the correct instance.
- **operation name** This is the name of the operation that raised by the process.
- **message data** Is simply the data of the message.

These data are necessaries to insert and retrieve events correctly.

BPEL to UI explanation

As can see in 4.4 there's a graphical representation of the components involved I this communication. The goal of this flow is to permit at the messages of the process to be read by the components. Talking about the artifacts, we want start to explain them from the bottom to the top. The first that we encounter is the *SOAP-UI Interface*, this is a receiver for the messages of the process. When a message arrives, this will be read by our component, and then, it will be inserted into the *persist event buffer*. In this

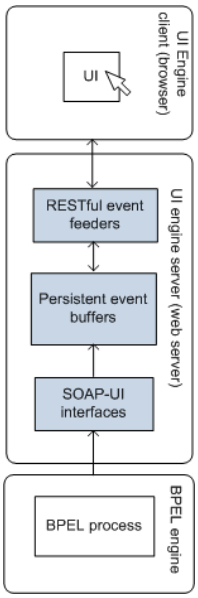

Figure 4.4: BPEL to UI communication

way, the operation data, are stored and can be read, in a second moment, by the UI component. The third element, the *RESTful event feeder* is a web service invoked by the client component when a polling operation is done. The service makes a request over the event buffer, using the parameters passed by the user element, if some events are present they are taken and replied to the component that can show them.

4.5.4 Endpoints of the services

Since now we explained how the communications works, which means the way that they follow from process to a component and vice versa. There's a question that we don't have explained yet, it is: how the BPEL can communicates to the SOAP-UI Interface and receive message from REST-Ful JSON-SOAP adapter? We know that the components are described by WSDL-UI files and they are used inside the BPEL-UI process like a web service. This WSDL-UI document is a standard WSDL document plus some extension, the point is: they are WSDL documents. When a WSDL document describes a canonic web service it usually contains some endpoints; an *endpoint* can be seen as the real location where the service exposes its operations and events, in other words: where the message has to be sent to perform a call, or also, the location where the messages are received by the service. Now, also our WSDL-UI contains these endpoints, one for receive messages and one for send messages, and its values are assigned by our system, these values are: the URI of each services. In the figure 4.5

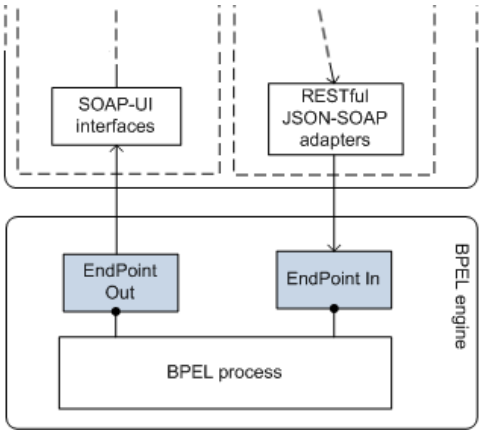

Figure 4.5: Endpoints for a UI

there's a graphical explanation about how the endpoints of a WSDL-UI (or WSDL at the end) are setup. As is possible to see in the figure there are two endpoints, one for each communication flow, incomings messages and outcomes messages.

4.6 Single and multiple instances

The solution presented here above is quite simple, but has some challenges problems that we have to explain, these challenges are due especially by some requirements and by the uses of the application that can be do. First of all, the UI components used can be reused more than one time in a single application, this means in example that a input box can appear in more than one page of a single process, or also more time in a single page. The second one problem, that's more interesting, and trick, as solution, regarding how to manage multiple instances of applications, especially: how the messages can be delivered right among the different instances. Here we explain also how we solved these problems, and how our solution introduces a new idea, and new features, over the BPEL language. Before proceeding the explanation there are some concepts and definitions that is useful to introduce.

- **Application**: with this term we want to describe the whole process, we use application to describe a process that is a BPEL-UI document: a BPEL process plus the UIs applications and pages.
- **Component**: this is an UI component, it's described by a WSDL-UI document. In the BPEL-UI language this means: there will be at least one port type that is linked to this WSDL-UI file. If more than one port type is linked to a WSDL-UI file we have more instances of the same component.

4.6.1 Single and multiple instances of Components

Sometimes there's a necessitation to reuse the same component more than one time, it can be because there are more users that needs it, or a single actor needs two, o more time, the same object. In the BPEL-UI process this is described by multiple partner links that point to the same WSDL-UI's file. In this way, in the BPEL process each instance is a different partner link, so the diversification of instances has already done in the code of the process. The problem might arise when we create the service and then we have to send the message at the right UI. The solution is simple and powerful, we create artifacts (the services) for each instance, in this way we trade each instance like a different component. If we have two instances of a UI component we will have (in theory): two RESTFul JSON-SOAP adapters, two SOAP-UI Interface, two RESTful event feeder and two persistent event buffer. As for the single instance, also in multiple instances of UI components, there will be the managing of endpoints. We create two endpoints for each instance, so in the single instances we will have two different endpoints, if we have two instances we will have four endpoints, for tree instances will be six endpoints and so on. In this way we are able to grab all the communications. Probably now a carful reader can raise a question: if there is more than one instance of the same application, how the system can deliver the message from a client to the right instance of the BPEL process? And how the UI can retrieve the message for the right instance for the user? We are going to answer to these questions.

4.6.2 Single and multiple instances of Applications

We have talked about single and multiple instances of components, but there's also to manage the multi instances of application. Normally each application, written in any languages, can be executed by more than one users on the same machine at the same time. Especially on the web, the same website is visited normally by more than one person. Also our system has to manage different instances of an application.

The developer of BPEL language has already thought about the problem of multiple instances of a single process, especially when the process is long-running with exchanges of messages with the partner (this introduce asynchronicity in the process). They have introduced, in the language, a mechanism called *correlation*. The correlation mechanism helps to route messages to appropriate process instances. It's based on the message involved in the communication, when the correlation is necessary a field of the message is used as key, as identifier value of an application. This key is matched with a value setup at the startup of the conversation, in this way the BPEL Engine is able to forward the messages at the right instance of the

BPEL process. This property has to be setup every time the conversation is asynchronous.

In our system we reused this idea for many reasons, the main is: the idea is powerful and simply to implement, and also the BPEL's correlation has to be implemented anyway, so the developer has to uses a field of a message to perform this mechanism. Our UI components contains in each message a field that is created only to be used as correlation, both for the BPEL's correlation (but is not mandatory to use this field for correlation) and also for our system's correlation. Each message that comes into the process will contains this field set by the UIEngine Client; on the other side, the developer has to setup this value for each messages that outcomes from the process, using the value of a previous one incoming message of the same flow. This value is used by our system in the BPEL to UI communication. When a message is sent by the BPEL process to the *SOAP-UI Interface* we need to know the value of the application's instance, this value is insert as field of the event during the insertion in the event buffer. In other words, when we insert in the *persist event buffer* the event, we also insert the key of the instance (see the 4.5.3 the field application instance key). The component when wants to perform a polling, request has also to pass the application instance key at the *RESTful event feeder*, thus, our feeder can search, using the value received, the event message. Done in this way multiple instances can simply managed and the messages are right delivered in the BPEL to UI communication.
In the UI to BPEL communication the problem doesn't exists, or better, is give almost for free. The managing of instances is done by the BPEL engine and not by our system. We have only to provide the messages with the field used for the correlation, the key of application instance. The correlation done inside the BPEL process, which is based on the field of message used for our system correlation, makes the BPEL engine able to manage the correct instance of the process, this implies that the correct instance of the process receives the message. For the UI to UI communication there's no problem to distinguish instances, the UI component remains on the same browser so the client-side's framework has only to send the data from an component to another one.

The mechanism of the key for the application instance is used every time an application is started. If there's one or multiple instances, our system use the key to identify the right process and then the right UI on the right browser. The same things that BPEL languages does, the correlation has to be setup also if the process is only one.

4.6.3 Powerful consequences of this solution

The consequence of this idea is a very powerful feature that in BPEL languages environment is not always possible. The big advantage of this uses of the key allows reusing a stopped process each time the user wants, this implies that an application can be started and stopped more time. Usually a BPEL process standard starts, perform its job and then it die. We have introduces a sort of stateful process or a stateful component, they have a state that can be reproduced when user wants. E.g. if in a process are involved two actors, which need to collaborate: the first one insert some data, the second one has to check these data before the final approval by the first one user. In this scenario the first one insert data on Monday, the second one checks the application on Thursday and fills his choices about the data inserted on Monday. After that the first one actor can finishes his work. Normally these things are not possible to do in a BPEL language, or better, a solution like this need a BPEL process alive for a couple of day. Our solution, instead, take advantages from the data stored to have a sort of state of the application. Using the *application instance key* an application can be restarted even if the process is not alive. The process has no state, so one instance or another one is not a problem (if it is stopped, otherwise there can be some problem, but this is not the question). If the process is running, the correct instance of the BPEL process is taken by the engine (as explained before). The vision of a BPEL process can be changed respect once time, the flow can be constructed to manage events and sends operations, from one user to another one.

5

The MarcoFlow framework in practice

5.1 Introduction

In this chapter we are going to investigate about the creation of the runtime server side components, we will explain how they are generated and deployed: which data are needed to construct each services, which parameters, the user, has to fill in the configuration file and which are taken automatically. We also explain how each service works in practice, which data are inserted at compilation and which are the data taken at runtime. This section goes deeply in the detail about how the artifacts works, probably all these detail are useful for someone who wants to understand how our system generates all the artefacts, how they are made and how they work. Here, we add more detail about each component, we will explain better how many components are used, where they are, and how the process can communicates with the components in which form.
Before go on, we will do a fast recap about some notions presented previously, which are important to understand all the things that will be written in the next sections. A process contains the whole behaviour, this means: all the things, which are done by the application, are written in the process. It also contains the pages, which are the different web pages that will be available at the final users. The components (UI components) involved into the process can have more than one instance, and each instance can be placed in different pages, this means that the components can stay on different UI engines. In fact we will have more than one UI engines where deploy the pages and its components; instead, the BPEL engine, will be only one.

5.2 Structure of an application's configuration

First of all we start from the configuration of a BPEL-UI project. As said in the chapter 3 the BPEL language now contains some extra pieces of information, these data are used at compilation time to retrieve some knowledge useful for compilation and deployment. In the process, the data used, are written in *partner link* and in *pages/page*. Inside the partner link there are also other pieces of information, not all are used in the creation, the only one

used is the *page* field. This field is used to identify the web page where the component has to be placed. Identifying the page, we identify also the engine where the component's services have to be deployed. All these data are not enough for a correct creation, they are only name that identifies engine, we need also URI of these machine and some additional values. To achieve this goal, we introduce a configuration of the application, this file, contains all the additional pieces of information needed. In the process there's simply the name of the engine, in the configuration there's a field with the same name that will contains all the other data. In the configuration there will be also data about the BPEL engine, the name of the project and some other descriptions.

Listing 5.1: Example of configuration

```
1 <sysconf name="ApplicationName" description="Desctiption">
2  <bpel>http://www.unitn.it/bpel/bpelngine</bpel>
3  <uiEngines>
4   <uiEngine name="UIEngine-TN">
5    <url>http://www.unitn.it/ui/uiengine</url>
6   </uiEngine>
7   <uiEngine name="UIEngine-HW">
8    <url>http://www.huawei.com/ui/uiengine</url>
9   </uiEngine>
10  </uiEngines>
11 </sysconf>
```

In the listing 5.1, here above, there's an example of an applcation's configuration. The data stored are simply to understand (they are underlined), the first line contains the name of the application and its description (*name* and *description* at line 1). The second line, between the *bpel* tag, is the URL of the BPEL engine. In the *uiEngines* tag there's a list of engines, in this case they are two. The first one, line 4, in named *UIEngine-TN*, and its location is specified at line 5. The second one is named *UIEngine-HW* and its location is in the line 7. Those are all the data necessary to complete the whole knowledge in order to create and deploy the components. Now, we are going to introduce our compiler and its works.

5.3 Compilation at work

In this section we are going too deeply in the explanation about how the compilation works, we provide this explanation using some schemas and pseudo-codes. First of all we divided, conceptually, the compiler in three parts: (i) the *compiler initialization* that show the initialization of the data and the cyclical operations performed, (ii) the *generation of the services*, which show the construction of service and what data are used and where are putted (iii) the *deployment of the services*, which illustrate how the deployment is performed. In the following sections we will talk about the compilation regarding the services side part, we will not talk about the

creation of the user side part (pages, communication of UI components and so on). The explanation will be taken with the help of some schemas that give a better understanding about the effort done by the compiler.

5.3.1 Step 1: Compiler initialization

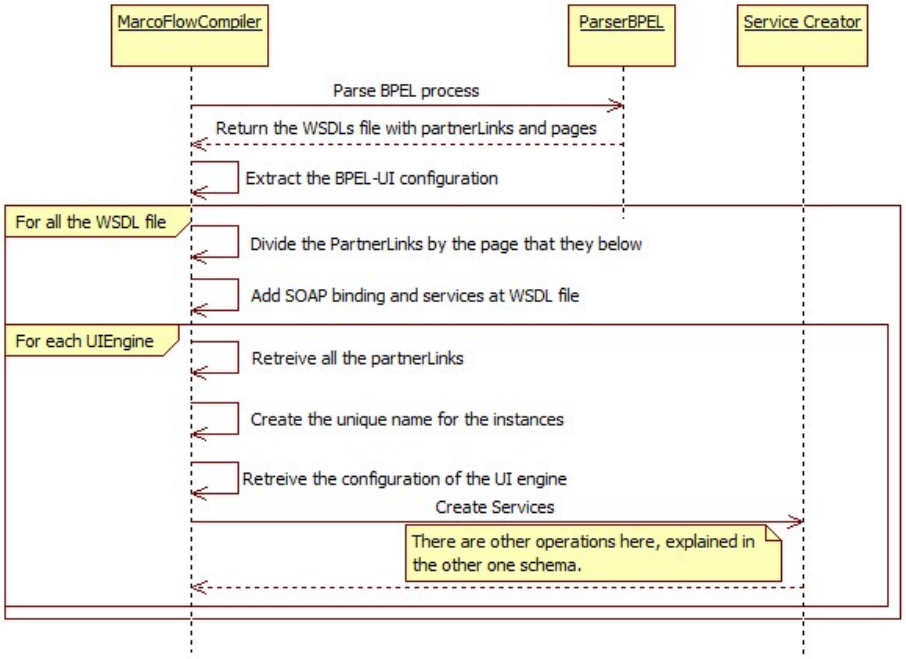

Figure 5.1: Compiler first step: initialization of data and cyclically operation over wsdl files and UI engines

In figure 5.1 there's a schema that shows the first step of the services' creation. Firstly, in the schema, we assumed that the BPEL-UI process, with all the related files, is present in a folder and the system already knows its location. As is possible to see in the sequence diagram, there are many steps in this part, now we are going to explain them better.
Parse the BPEL-UI process the compiler, here, retrieves some pieces of information from the BPEL-UI process. The data needed are: all the partner link-UI (that are links used in BPEL standard process which allows using the WSDL file associated) involved in the process with the associated WSDL-UI file. The compiler also provides, directly, the name of the associated UI engine for each partner links. In this way we have a list of WSDL-UI files, with location, a list of each partner links that are connected to a WSDL-UI file and also the list of UI engines for each partner link. The data that the compiler retrieve from the process are not enough, as said it needs some

extra pieces of information that are taken from the configuration (*Extract the BPEL-UI configuration*)
In this point the compiler has all the data its needs, thus it starts to iterate over the list of WSDL-UI files involved in the communication, and in the *for each WSDL-UI file* step it performs:

- Divided each partner links (*Divide the Partner links by the page that they below* by the UIEngine that they belong. We know that a component can have more than one instance and these instances can belong to different pages. The pages have to be deployed on different UIEngines, in other words, the engines can be different for each instance and we have to manage this. Dividing the partner link by UIEngine, we have all the instances of a component that has to be deployed on a specific UIEngine each time.

- The WSDL-UI file, as it is, doesn't contains information that are mandatory in a WSDL file for a service, in the *Add SOAP binding and services at WSDL's file* step we add the binding and services section at the file. The first section, the binding, maps the operations; this section can see as the interface for the web service. The second one section, the service, which is connected to the previous one, contains pieces of information about location of the services, in other words: the URL where the service can be invoked. With these modifications the WSDL-UI file has almost all the fields of a standard descriptor for web service, the only information that is not present is the port (the URL) of the service and that are add later after the compilation step.

Inside this cycle, the compiler has to create the services' file for each UI engine, as said, we always have at least one engine, but we can have more than one engine for the same WSDL-UI file, so we have to perform a cyclic operation over all the engines. In this step the compiler is working on a single WSDL-UI file, this file is taken by the main cycle.
Over the WSDL-UI file already taken, and for each UI engine of this file, it does:

- Select a specified engine, it *retrieve all the partner links* of to the WSDL-UI selected and which belongs to this UIEngine.

- For each partner link creates a unique name (*creates a unique name for the instances*). A partner link is an instance of the UI component, and any instance has to have a unique name.

- Finally there is the creation of services *Create Services*. This creation involves all the instances of the same component that belongs to the same UI engine.

The first step of the compilation finishes here, now we are going to talk about the creation of services' files.

5.3.2 Step 2: Creation of communication' services

The creation of services is probably the most interesting part to known, our solution for coordinating UI components and processes is based on a series of services that has the same skeleton. These services, however, needs to be fill in with the data of each component, because some parts of the code are specific for the component that they wraps.

In the chapter 4 we explained how the idea works, here we present two kind of communication for each component. In practice, instead, when we built the services we try to save space on the server, so we decide to build up services that are able to manage the communications for each instance of components. In some case, as will explain later, there will be only one service that manages communication for each instance of component inside an application. In example, if in our application we have (on the same engine obviously) three instances of the same component, we not always have a service's file for each instance, but we will have a service that is able to manage all these three instances. Anyhow, before start with the explanation, we have to keep in mind that the compiler sends to the creator of services some data, these are:

- The application name
- A list of all the component instances' name.
- The WSDL file's location.
- The configuration of the involved UIEngine, that is: name and URI of the machine.
- The configuration of the BPELEngine, which is the URI of the machine.

These data are obviously used in the creation of services, the first four are used inside the code, the last two are used in the creation of endpoints. In figure 5.2 there's a sequence diagram regarding how the compiler creates the services. Is high level, it not represents all the steps for the whole creation of services, the creation of files is tricky and its explanation needs a proper section. Anyway, the figure shows the logical flow of the compiler and it's useful to have a background of each step. The behaviour is simple, probably there are some parts that are not clear at the first time, we try to treat them in a deeply manner in order to make them understandable.
Anyhow, the compiler, here, does:

- *Retrieves from the WSDL-UI file all the receive operations.* These operations are used in the process to receive message from outside, in our case them are used to receive *events* from the UI component.

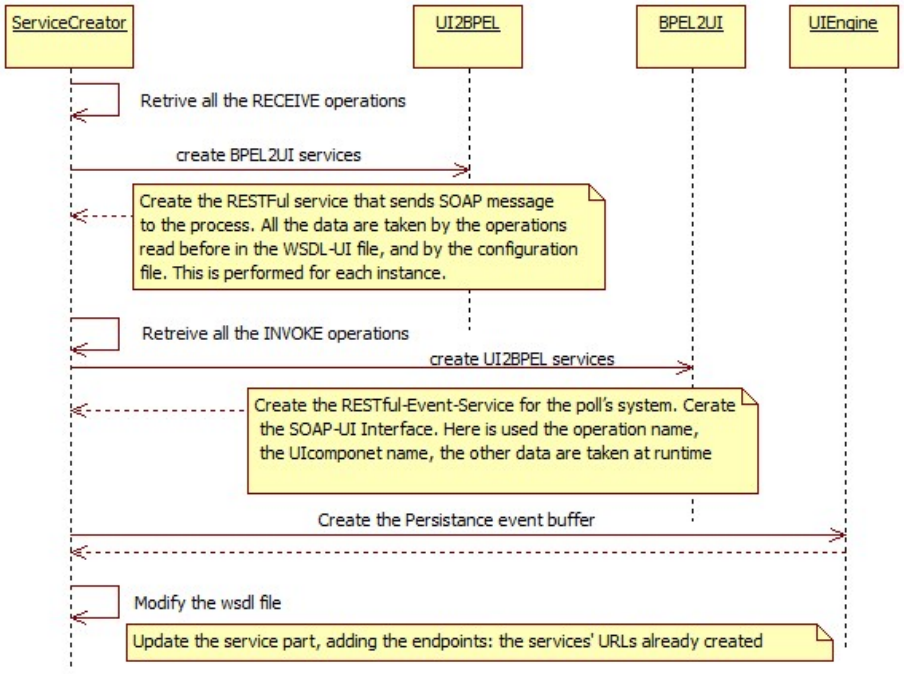

Figure 5.2: Compiler: creation of services' files

- Creates the UI2BPEL communication's services, which are the *RESTFul JSON adapters*.

- *Retrieves from the WSDL-UI file all the invoke operations*. These operation are used in the process to send messages outside, in our case them are used to send *operation* from the process to the component (an *operation* is a message received by the components, in the services environment the message is received by the *SOAP-UI Interface*) .

- Creates the BPEL2UI communication's services, that are the *SOAP-UI Interfaces* and the *RESTFul event feeders*.

- Finally, the last operation for complete the artefacts' generation creates the *persist event buffer* on the UIEngine. The creation is performed among the net, the compiler simply contacts a web service on the engine that creates the storage item on its machine.

- The last step of the compiler, regarding the modification of WSDL-UI file. It adds, at the file the right endpoints generated here above, in this way the file becomes a WSDL file. In this point of compilation, the compiler, has all the data needed to known the correct URI of the services. It places these values inside the corresponding service, as a

port of the WSDL file. In this way the descriptor of service has the whole data needed to map a web service, this means that it can be used as descriptor of standard web services in a running BPEL process.

In this paragraph we have talked, high-level, about the creation of the services, we haven't explained how these services are really created, in the next sections we try to avoid this gap and explain each step of the creation.

BPEL2UI communication's services creation

This communication is the first that we encountered during the creation, we have encountered it in the second step of the compiler (section above). As already explained previously, the services involved in this communication are three and they are: *RESTFul Event Feeder* , *SOAP-UI Interface* and the *Persistent event buffer*. The last component, and its creation, is already explained here above, so we don't trade it anymore. Now we talks about the creation of the two other artefacts.

Creation of a SOAP-UI Interface

This service is the endpoint used by the process for its out-coming messages. As said, these endpoints are specific for each instance, so we have to construct a service's file for each instance of component that has to be handled by the process. In order to achieve this goal, we used a technology for services that allows placing these services in a specific place without configurations. This is good for us, because we can have endpoint that cans be simply distinct by the application's name and the instance's name. A typically endpoint for a SOAP-UI Interface is something like this: *$UIEngineURL$/services/$ApplicationName$/$InstanceName$*. Between the "$" there are some marker places that identifie where some data are placed, the name are self-explanatory.
During the creation of this service the data that has to be inserted are only two, and are: (i) *application's name* (ii) *instance's name*. These two values cannot be retrieved at runtime, the explanation is simple: there are data that the runtime environment doesn't know and the message doesn't contain, so no one can provides them at runtime. The only one place where we have these data is in the compilation, in fact, these two data comes from the configuration file and from the generation done by the compiler. These two fields, used at runtime, permits to know: where the message, arrived from the process, has to be placed into the *persistent event buffer*. The first datum, in fact, correspond to the name of the table, the second one is used as identifier for the correct instance of the component among the others instances.

Creation of a RESTFul Event Feeder

As far as we know, the *RESTFul Event Feeder* is used in the pooling sys-

tem, it has to search the data from the *Persistent event buffer* and reply them to the UI component. In the creation step, this service doesn't need a lot of pieces of information to be constructed. It needs only: the *application's name* and the*UI component's name*, these two values are used to create the service in a way that can be accessible from an URL like this: *$UIEngineURL$/BPEL2UI/$ApplicationName$/$Compon- entName$*. Now probably, the reader can ask why we don't use the same method saw in the creation of SOAP-UI interface. The answer can be found in choice made: when we built these services, that are RESTFul services, we have decided to use a "standard" technology that has more feature respect other solutions, the method choosen has some capabilities, like the simply managing of data, that are very useful for our propose. Turning back to creation's description, the other data that lack here are taken directly at runtime in the request (the message) sent by the UI's polling system. The information inserted at compilation, as said, are used to extract the data, they follow the same paradigm of the *SOAP-UI Interface*. Done in this way, taken the most parts of the service code at runtime, and have not the strict necessitation of a single endpoint we can construct a service that is able to manage all the instance of a component. This means: we have a unique service that manages all the instances of component, and we don't have a service for each instance like in the *SOAP-UI Interface*. This solution saves space and effort for the engine.

UI2BPEL communication services creation

This communication is different from the previous one: is stateless in some sense, its data pass among a web service that has, as goal, the calls of the process, so nothing is stored in the database. The services involved in the communication, named *RESTFul JSON adapters* has to receive each JSON message from the associated components, translate this message in a SOAP, well defined, message and send it to the correct endpoint. This service is probably more complex respect the two saw before, it has more logic inside. The message, which comes from the UI component contains, with the data of the message, also other pieces of information that has to be used to create the SOAP message. In this case, the big amounts of data are taken at the compilation time.
First of all, in a communication to BPEL process, we need the endpoints of the process, and we need the endpoint, and they have to be able to receive the operations for a determinate component. These URL are made in a way that permits to recognize and construct them by a fast and simply method. They are simply composed by a prefix, which is the URL of the BPEL engine, the second part is the name of the application and finally, the last parameter, is the name constructed by the system that identifies uniquely the instance of the component, i.e. an endpoint can be *$BPE-*

LEngineURL$/$ApplicationName$/$InstanceName$. The first two values are taken by the configuration. The last part of the URI, which is made by our system, is used also inside the UI client framework as name of the polling component. In this way, also the pooler of the UI component, when performs the polling call, can sends this data at the service. We decided to receive this value at runtime, instead to fill in the service at composition, because we want to manage all the instances of a component inside a unique service. Sending the name, we can simply change the endpoint at runtime, which means: we are able to manage more instances of component in one file (the relation in 1-N, 1 service N instances).
There are also other data inserted at compilation, these are: (i) operation's name, (ii) operation part's name, (iii) UI component's name and (iv) application's name. The first two are used to create the correct SOAP message, we provide a management for each *event* of the WSDL-UI file, so we have to distinguish, depending on the operation name, which message has to be created. The operation part name is used inside the message, the rules of the WSDL encoding need this value inside the message. The last two values are used to permits the placement of the service in a correct folder on the engine and makes this service available on an already known URI (the same thing that happens in the creation of *RESTFul Event Feeder*).

5.3.3 Step 3: Deployment of services

We talked about these services, and engine, but we don't talk how they are deployed on the right machine. As probably is already known our system allows having a distributed environment: there are different UI engines that are running in different location. Obviously, in a distributed scenario, the engines communicate to the BPEL engine among the net. The first thing to do, in a scenario like this, is to deploy the files on the right engine. We provide a solution, which is based on web service. This web service is present on the UI engine and has as goal to deploy, in the correct place, the services' files received from our compiler. In practice, it receives a compressed file that contains all the services; these files are in a text format, so it has to compile and move them to the correct place. This solution permits to have a scalable system that doesn't need effort by the user at deployment. As said, a process to be runnable needs almost a BPEL process that performs the logics. Also the process has to be deployed to the correct engine, this engine can be on the same machine of the compiler or it can be dislocated in a different place. In other words, we have to provide a method to deploy process on distributed engine, to achieve the goal we use the same method said before: we built a dedicated web application that has to be putted on the server of the BPEL engine. This application allows deploying BPEL process and other necessary file, simply sending the data to the specific web service, this service receives the file and put them in the correct folder.

The distributed scenario can raise a question about where the services are located. As explained before, during the creation of services, we explained that the compiler knows the endpoints, and after the creation it updates the WSDL file inserting these values. In this way, the WSDLs files will contains the correct endpoints, and the BPEL process, manages them like standard web services. In substance a half of work about the distributed environment is given by the existing technology in the BPEL languages, we have only to provide the right service in the right place, but this is no so trivial.

We have talked about the deployment in theory, we said something about how it works, but now we want to be more precise in his functionality, in fact, the third step of the compiler is *the deployment*. In figure 5.3 there's

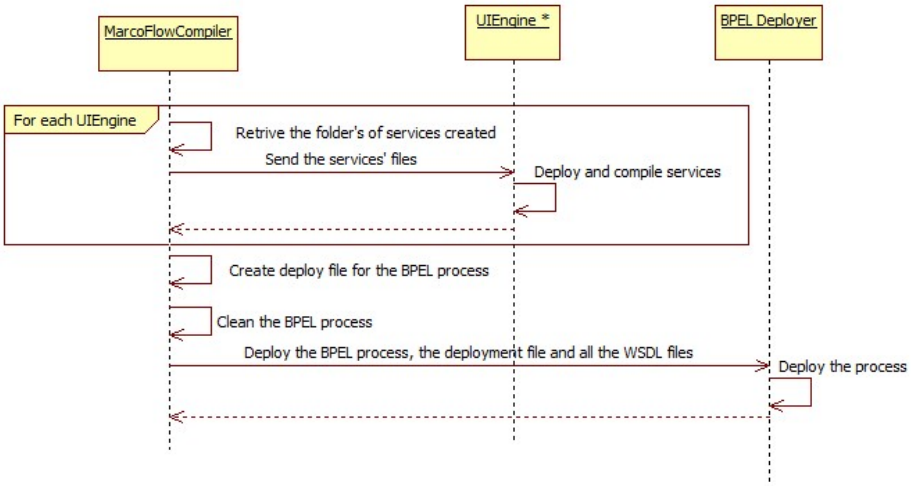

Figure 5.3: Compiler: how the deployment works

the sequence diagram of the deployment part. When the files were created, they are placed into a specific folder that identifies each UI engine and has as root a folder with the name of the application. (i.e. if the engine name is *engine-TN* and the application is *find-it* the services will be placed in /find-it/engine-TN/). In this manner the compiler, when has to deploy the file, can simply search for each UI engine's folder inside the application's folder. It will take the files inside and they will be sent at the correct engine. When the file are received by the engine, they are compiled and placed in the correct folder, in this way they are available and ready for the use.

The services aren't the only files that need a deployment, also the BPEL process need to be placed on its engine. The BPEL engine, instead the UI engine, need also a deployment file that contains data about which services has to be exposed and which services are used to send the data. The compiler

also creates this file automatically, parsing the information presents in each WSDL-UIs files. Once created this file is putted inside the folder of the BPEL process and sent to the engine. Thus, the engine can deploy and run the process.

5.4 Services at runtime

We want to spend some words about the behaviour of our services at runtime, since now we have talked about how they are constructed and what they have to do basically, but we didn't trade about their runtime behaviour. In these subsections we provide explanation using sometimes a listing of code, the code inserted is a java style pseudo code that wants to shows the behaviour of the services step by step.

BPEL2UI services execution

Here we explain how the BPEL2UI communication is managed at runtime. In this communication there's a message that comes out from the process and goes to the UI component. All the idea behind each component is explained in the previous section (see section 5.3.2).

SOAP-UI Interface The goal here is: receive a SOAP message, extract some information and then store the converted message in the event buffer.

Listing 5.2: Snippet of a SOAP-UI Interface

```
1  //Init
2  String ID_FIELD = "id_correlation";
3  String name = $name$;
4  String instance-name = $instance-name$;
5  //execution
6  SOAPMessage soapMessage = receiver.getMessage();
7  soapMessage.clean();
8  XMLDocument xmlDoc = soapMessage.getBody();
9  String operation-name = xmlDoc.getOperationName();
10 String application-instance-key = xmlDoc.getApplicationInstanceKey
       ();
11 JSONObject json-message = xmlDoc.convertToJSON();
12 //insertion
13 eventbuffer.insert(name, application-instance-key,instance-name,
       operation-name,json-message);
```

In the listing 5.2 there's the pseudo code of the SOAP-UI Interface's, and there are two value, underlined, that are the value inserted in at the compilation time. The first one field, named *ID-FIELD*, is used as discriminator the name of the application. How is understandable by the code, when a SOAP message is received, the first step is to remove all the SOAP stuff, after this it takes only the body of the message that is an XML document. This document contains data, some of them (like ID-FIELD) are taken from the

document and removed, the other still remain in the doc and are converted in a JSON Object. Finally in the last line there's the insertion of the event (*eventbuffer.insert*), it uses the data inserted at compilation (line 2 and 3) and the data taken at runtime (line 10-11-12, which are the id-application, operation-name, json-message) to store the value in the *Persist event buffer*.

RESTFul Event Feeders
This component performs the polling system.

Listing 5.3: Snippet of a RESTFul Event Feeders

```
1 Service $ui-name$ {
2  String NAME = $ui-name$;
3  String OPNAME = "opname";
4  String INDEX = "index";
5  String INSTANCE-NAME = "instance";
6 ..
7  Operation JSONObject get(String application-instance-key,
       JSONObject parameters) {
8    String operation = parameters.getString(OPNAME);
9    String index = parameters.getString(INDEX);
10   String instance-name = parameters.getString(INSTANCE-NAME);
11   return eventbuffer.read(NAME, application-instance-key,instance-
         name, operation, index);
12 }
13 }
```

In the listing 5.3 here above there's the pseudo code of the *RESTFul Event Feeders* . Also in this service, like the previous one, some data has to fill in at compilation (underlined), in this case the value is the service name, which is given by the compiler. The data on the line 3 and 5 are static field, they identify uniquely some field of the message, and they are inserted in this way to have a better maintenance of the services' code for the future. From the code we can also see that this service has a method named *get* (line 7), which returns a JSONObject, and has as parameters a JSONObject and a string. The return object is the event. The input parameters are two, one is the*application-instance-key* that is used in the search of events, the other one parameter is an object that contains all the other data needed in the research of event, these fields are:

- The name of the instance of the component (not the component, but the instance of the component)
- Operation's name, which is the name of the operation event stored
- The index value that is used to retrieve the index-th event in the buffer

In the last line there's the search of the event in the buffer, as is shown there are many data used. The first one (name) is given at compilation, the second is given as parameters and is the application instance key that identifies the application instance, the other are taken by the JSONObject

and identifies, in order: the instance name (so the instance of the component), the operation that has to be search (the operation for the UI), and the index. We never mentioned the index value before, this value is used to retrieve the index-th event from the buffer, i.e. if there are five events, and the index is three, the service return the third events found.

UI2BPEL services execution

The UI2BPEL communication contains only one service, which is the *RESTFul JSON-SOAP adapter* that constructs a SOAP message starting from the JSON data received by the UI component.

Listing 5.4: Snippet of a RESTFul JSON-SOAP adapter

```
1 Service $ui-name$ {
2  Operation send(String instanceName, JSONObject action) {
3   //retrive opeartion name
4   String operationName = (String) action.get("opname");
5   //each if is an operation wrapped
6   if (a.equals($operation-Name$)) {
7    //retrive parameters for the message
8    JSONObject parameters = action.getJSONObject("pars");
9    //extract operation
10   String operation-PartsNs = $operation-PartsNs$;
11   String endpoint = $UIEngineURL$ + $applicationName$ /+
         instanceName;
12   String operation-PartsName = $operation-PartsName$;
13   sender.createAndSendMessage(operation-PartsNs, operation-
         PartsName, parameters, endpoint);
14  }
15  ...
16  //if for each operation
17  ...
18 }
19 }
```

In the snippet 5.4 there's a description for the *RESTFul JSON-SOAP adapter*. The underlined string highlight the data inserted during the compilation, these values are:

- The name of the component (line 1) is used also to define the service's name.
- Operation-Name (line 9) is used as discriminant in the operation's message selection. Each operation, in facts, needs a proper SOAP message that contains differently data.
- Operation part namespaces (line 10), is used in the construction of SOAP message, this message has to contains all the namespaces used.
- UIEngineURL (line 11) is used to construct the final endpoint.
- ApplicationName (line 11) also this value is used in the creation of the endpoint.

- Operation part name (line 12) that is used as root inside the body of the SOAP message. The WSDL encoding that we used necessity of this value instead the name of the operation.

The values here above are inserted at compilation, but some data are taken during the execution, these value are passed in two parameters, a string (*instancename*) that is the name of the instance (which is a partner link in the BPEL) that the components wants to call, and a JSONObject (*action*) that contains a series of data. The data present in the last parameter are:

- *Operation name* (line 4), this information is used to search which operation is called by the UI component. The service, once made, contains all the operation of the component (there will e a series of *if* clauses for each operation), and then, at runtime, it needs to find the right operation and the corresponding message to create. When a runtime operation's name is equal with an operation's name inserted at compilation the service knows which kind of data has to use to create the SOAP message.

- parameters (line 9) these data are the body of the SOAP message, they contains all the value that has to be passed to the process.

- instanceName (line 11), which is the name of the instance that performs the call, and also the last value of the endpoint URL. Each instance of components corresponds to a determinate partner link and each partner link has an endpoint that listen for incoming operation. Using this value we are able to contact the correct endpoint, in fact, this value is placed at the end of the final endpoint.

The only parameter that we didn't have directly in this service is the *application instance key* this value is provided inside the *parameters* so it is inserted directly in the SOAP message as field of it. In this manner the process receives the datum and it is able to use it. Finally, with all these data, the services create the message and send it to the correct endpoint, in this way a message is received by the process that performs its flow.

6 Working scenario

6.1 Introduction

In the section 1.3 we presented the reference scenario for this project, now here, we are going to present the same scenario in practice, using it as demonstration of the power of MarcoFlow. We will present both the scenarios (single and multiple users), we will also explain - high level - how the process has to be created and how the components are inserted. We will give a deeply explanation about the things pertaining to this thesis, so we will explain better the steps regarding the creation of their services, the deployment and their execution.

6.2 Process and Components

The first thing to explain is how the process is made and also how, and which, are the components used. This explanation will be done high level, only to give at the reader a basic knowledge about the creation of the process.
The figures of components and processes are directly extrapolated from the BPEL-UI editor this gives us the opportunity to present the UI object in a way more closed to the developer view.

6.2.1 UIs components used

The list of the components was described in the introduction of the scenario (section 1.3), here we want to go a little bit in detail showing the operations and the messages that each one component can do and provide. The components are principally three: (i) *Search form* which contains the input box for the search operation (ii) *List* it's a simple list for the answers (iii) *Google maps* this component is trivial, is the Google map. In the explanation we show some figure taken directly by the editor, the things to view are the *name of the operation* (near the gear icon) and if it is an operation with *input* (that's called *operation*) or *output* (in this case is an *event*). In detail they are:

Component: Search form

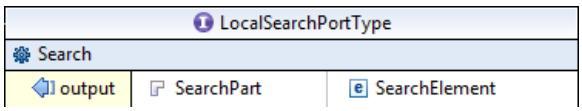

Figure 6.1: Operations of the Search Form component

Figure 6.1 shows the operations and the events of the *search form* component. This component has only one *event* that is: *Search*. This event sends the request for the search operation.

Component: List

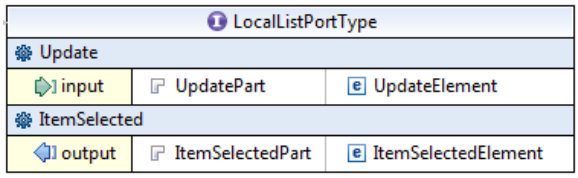

Figure 6.2: Operations of the List component

Figure 6.2 shows operations and events of the *list* component. In this component we have an *operation* that is *update* whose goal is to update the list on the page; and an event, *item selected* used to intercept the item selected and then send the coordinate to the map.

Component: Google Map

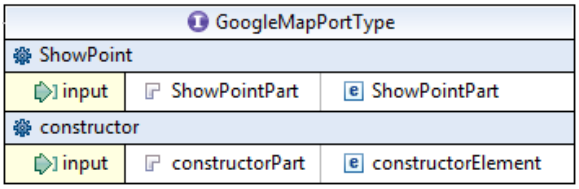

Figure 6.3: Operation of the Google Map component

Figure 6.3 shows the operations of the *google map* component. In this case there are no events, this component is used only to display the item selected by the user. Instead, there are two operations, one is for the component itself, named *constructor* it's used to instantiate the maps components inside the page. The second one, *showPoint* that is used to show a point on the map.

6.2.2 Design of a BPEL-UI process

After shown the components, we are going to introduce the process that manages these components. In this section we show only the high level process with some explanation, we won't go in detail about all the specification of the whole code, we will provide some figures of the processes and a minimum pieces of code to understand their uses.

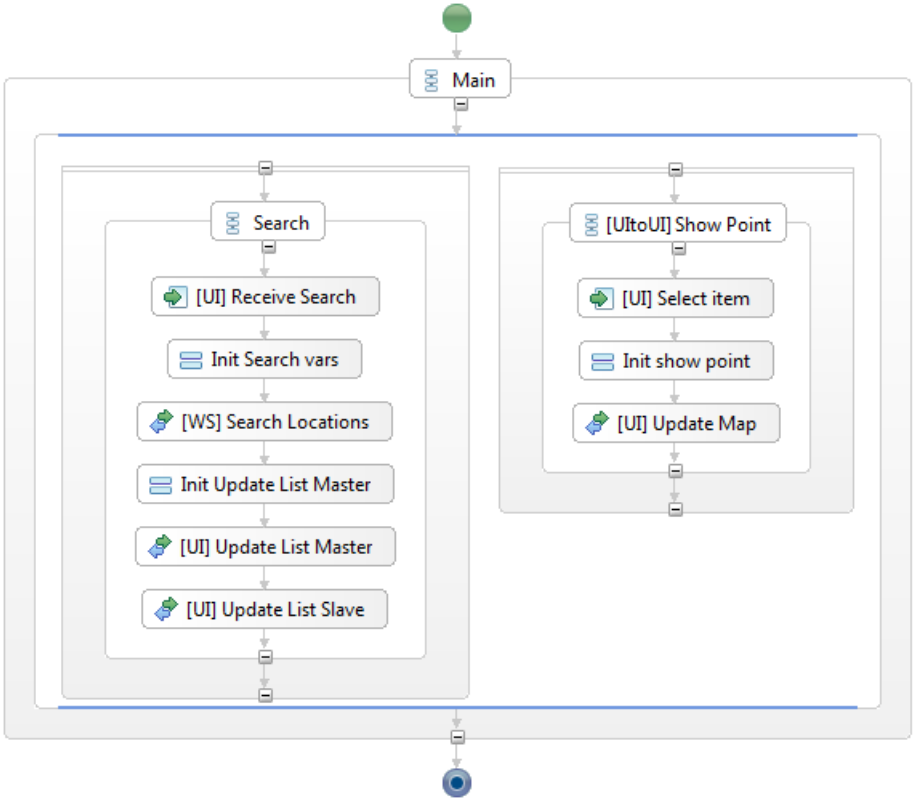

Figure 6.4: FindIT applicaion, single user scenario

Figure 6.4 show the BPEL-UI process for the FindIT single user application. The process looks like a standard BPEL process, the differences are in the code (that we show after). In the design of the process we use some textual annotation only for a better visual understanding of the process. The operations with the prefix [UI] are used to annotate the iteration with the UI component, the other one, with the prefix [WS] for the communication to external web services. As visible, the process contains two flows that can be executed in parallel, one named search, and the other one [UItoUI] Show point. The goal of the first is: receive data from the Search Form component ([UI] receive search), search locations on the web service (WS] Search Loca-

tions) and update the list component in the user's page with the locations (WS] Update List). The other one flow, as the name suggest is a UI to UI communication, this means that this communication will be performed on the browser, without interaction with the process. The goal of the second flow is: receive the item selected by the user in the list component ([UI] Selected item) show the item in the Google maps ([UI] Update Map).

The multi users process introduces some extra flows and adds some other interactions in existing flows. Figure 6.5 shows the process of the FindIT

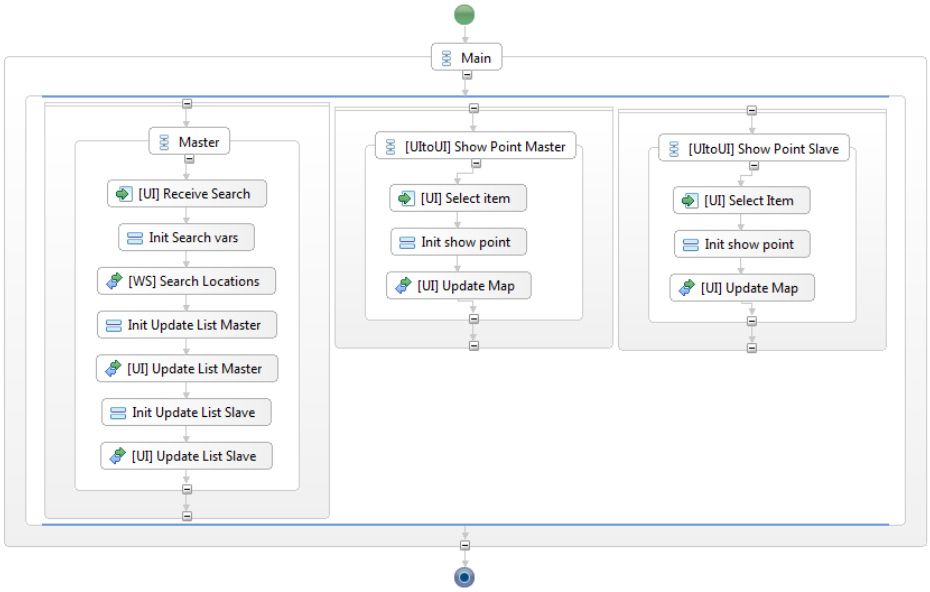

Figure 6.5: FindIT application, multi user scenario

multi user application. Is similar to the single user application, but here there's another one UI to UI flow for the slave page (on the right) and another one interaction in the search of the locations (on the bottom of the first flow, at left, there's also *[UI] Update List slave*). The behavior is similar to the previous one process, in this case there's a managing of the item selected for the both actors (so for the both pages) and also forward of the data founded at the slave page (first flow).

The news introduced in the model, described by BPEL-UI language, as said, are in the code of the process, in the listing 6.1 there's a snippet of the new codes introduced by our definition.

Listing 6.1: Snippet of BPEL-UI process

```
1 <pages>
```

```
 <page name="master" templateURL="http://www.unitn.it/BPELLayout/
     masterLayout.html" uiEngineName="FindITEngine" actorName="SteS
     " description="the master container" isStartPage="true" />
 ...
</pages>
 ...
<bpel:partnerLinks>
 ...
 <bpel:partnerLink name="SearchForm" partnerLinkType="
     ns1:SearchForm" myRole="receive" partnerRole="invoke"
     isUiComponent="yes" pageName="master" placeholderName="
     marcoFlow-left"/>
 ...
</bpel:partnerLinks>
```

The new things are mainly two, the *pages* item that contains a list of *page* items; the item page contains the data useful to identify the page, the layout the actor and the engine to use. The other one news is in the partner link tag, we add some new field to identify the UI component, the page where they have to be placed and some other things. In the snippet there are some field underlined, these field are those used in the generation and deployment of services. First of all, starting from the partnerLink, we find out the UI components and the page where they belong. Second, from the pages item we extract the information about the engine where these services have to be deployed; there's a correspondence between the *pageName* in the partner link and the field *name* of the item *page*.

Listing 6.2: Snippet of the multiple instances of a component in the BPEL-UI process

```

<bpel:partnerLinks>
 ...
 <bpel:partnerLink name="ListMaster" partnerLinkType="ns2:List"
     myRole="receive" partnerRole="invoke" isUiComponent="yes"
     pageName="master" placeholderName="marcoFlow-left"/>
 <bpel:partnerLink name="ListSlave" partnerLinkType="ns2:List"
     myRole="receive" partnerRole="invoke" isUiComponent="yes"
     pageName="slave" placeholderName="marcoFlow-left"/>
 ...
</bpel:partnerLinks>
```

The last thing to show about the process regards how we describe the reuse of the same component, that's a multiple instance of the same component. As is visible in the listing 6.2 there are two partner links that point at the same file (they point to the same partnerLinkType, but this is a reference to the WSDL-UI file). This declaration identifies a multiple instance of the component *List* in this case one of them will go to the master page and the other one to the slave page. This snippet will be useful to understand the things about the generation of the services in a multiple component instances scenario.

After the design, the BPEL-UI process and all the files of the components are ready to be deploy, the last thing that lack is the *configuration file*.

Listing 6.3: Configuration for the FindIT application

```
<sysconf name="FindIT" description="FindIT scenario">
 <bpel>http://www.unitn.it/BPEL4UI/DeployerBPEL</bpel>
 <uiEngines>
  <uiEngine name="FindITEngine">
   <url>http://www.unitn.it/BPEL4UI/UIEngineServer</url>
  </uiEngine>
 </uiEngines>
</sysconf>
```

In this case we have only one engine, named *FIndITEngine* that contains the URL of the machine where the engine is deployed. The firs line, instead, describes the process with name and description, the second line contains information about the BPEL engine used in this application.

6.3 Creation of the middleware

In the past section we saw how the process is generated and what its behavior is; in this section we trade the passage from the description to the generation of all the components of the middleware.
As explained, in the single user scenario there are almost all the basic communications: (i) component to process, which is the communication from the Search Form to the process, (ii) process to component, which is the update list by the process and (iii) component to component, which is the item selected to the show point into the map. The first two (i) and (ii) has to pass through the framework, so here we explain the steps involved in the creation of the artifacts in practice and how they are used in these communications.

6.3.1 Management of the communication

UI2BPEL communication

In this scenario, this communication, regards the event sends by the Search Form to the BPEL process. As explained in the previous chapters, this communication is handled by only one service named *RESTFul JSON-SOAP adapter*. This component wraps all the events that are generated by the component; in the Search Form case the event is *Search*. The services, is able to manage all the events in a unique service.

Listing 6.4: RESTFul JSON-SOAP adapter for Search Form component in the FindIT application

```
Service SearchForm {
 Operation send(String instanceName, JSONObject action) {
  //retrive opeartion name
  String operationName = (String) action.get("opname");
```

```
5   //each if is an operation wrapped
6   if (a.equals("search")) {
7    //retrive parameters for the message
8    JSONObject parameters = action.getJSONObject("pars");
9    //extract operation
10   String operation-PartsNs = "www.unitn.it/WSDLUI/SearchForm";
11   String endpoint =
         "http://www.unitn.it/BPEL4UI/UIEngineServer/FindIT/"+
         instanceName;
12   String operation-PartsName = "SearchElement";
13   sender.createAndSendMessage(operation-PartsNs, operation-
         PartsName, parameters, endpoint);
14  }
15 }
16 }
```

In the listing 6.4 there's an idea about how is the service's code, as is possible to see respect the skeleton presented in listing 5.4 all the compilation fields are filled in with real values, the parameter of the method are passed at runtime and used to manage the correct instance of the component via the endpoints coordination.

BPEL2UI communication

The process to component communication, in this case, regards only one component, the *list*, the artifacs generated for the wrap are three. Starting from the bottom, and following the flow of the message, the first service encountered is the *SOAP-UI Interface*.

The structure of the *SOAP-UI Interface* created is not too much different respect the skeleton.

Listing 6.5: SOAP UI interface for Search Form component in the FindIT application

```
1 //Init
2 String ID_FIELD = "id_correlation";
3 String name = "List";
4 String instance-name = "List_master";
5 //execution
6 SOAPMessage soapMessage = receiver.getMessage();
7 soapMessage.clean();
8 XMLDocument xmlDoc = soapMessage.getBody();
9 String operation-name = xmlDoc.getOperationName();
10 String application-instance-key = xmlDoc.getApplicationInstanceKey
       ();
11 JSONObject json-message = xmlDoc.convertToJSON();
12 //insertion
13 eventbuffer.insert(name, application-instance-key,instance-name,
       operation-name,json-message);
```

Respect the code saw in the listing 5.2 the code here above (listing 6.5) contains the name of the component and the instance name, as is possible to see this instance name is given by the name of the partner link plus the

page where it below. A services created in this way is able to intercept all the communication from the process, and insert them into the buffer in a correct way.

The second artifact that we have in this communication is the *persist event buffer*, it has the same structure for all the components, the only thing that changes is the name of the buffer, that is the name of the component, in the exaple is *List*.

The last item, the third, is the *Restful Event feeders*, its goal is to provide the data at the request done by the polling system. Also the structure of this component is similar to the skeleton (listing 5.3).

Listing 6.6: Restful Event feeders for Search Form component in the FindIT application

```
1 Service List{
2  String NAME = "List";
3  String OPNAME = "opname";
4  String INDEX = "index";
5  String INSTANCE-NAME = "instance";
6 ..
7  Operation JSONObject get(String application-instance-key,
       JSONObject parameters) {
8    String operation = parameters.getString(OPNAME);
9    String index = parameters.getString(INDEX);
10   String instance-name = parameters.getString(INSTANCE-NAME);
11   return eventbuffer.read(NAME, application-instance-key,instance-
         name, operation, index);
12 }
13 }
```

Listing 6.6 show the services after the creation. The data filled in is only one and it is the name of the component; the rest of the values are taken at runtime.
These three services are able to manage the BPEL2UI communication as well.

6.3.2 Multiple instance of a component

As explained in the listing 6.2 and in its explanation, there's a possibility to reuse the same component more than one time. In the multi user scenario, we have this peculiarity with the *list* component. This component has only the BPEL2UI communication and the services are presented above. In a multiple instances of components, the services still remain almost the same. The unique difference regards the*SOAP UI Interface* service, in this case, these services has to be unique as files, this because they are the endpoints for the outcomes messages. The message that the process sends don't contains identification about the final UI component, the discrimination has to be done via the endpoints, so we need a specific endpoint for each component.

The only thing that changes, among the files of each instance, is the *instance-name*. In this scenario, these value will be *ListMaster_master* (partner link name plus page) for the instance of the master page, and the other one will be *ListSlave_slave* for the other instance.
In the case of a UI2BPEL communication, the file will remain the same if the instances is one or multi. As explained in the creation chapter, the instance's name of the component is always taken at runtime, this permits a flexibility of the services that is able to manage multiple instances of a component without modification of the code.

6.3.3 Deployment of services and process

In this scenario, the services are deployed on a single machine, accordingly with the configuration. Once created, the services are placed in a folder named FindIT. After this, the folder is zipped and sent to the UI engine deplorer service. Once received, these services are compiled and made available among the net.

The deployment of the BPEL-UI process is more complex, the main problem is the absence of the deployment's configuration. This file is generated by our system and it contains all the information about what port, of each service, has to be used by a partner link. The cleaver solution, used in our compiler, creates in each service a port for each instance of the component, in the case of the *list* item, we will have two incoming endpoints and two for the outcomes endpoints. Thes values are assigned by the compiler following the place where the services are deployed or exposed.

Listing 6.7: Endpoints specification for the list component

```
1 ..
2 <wsdl:service name="ListIncoming">
3 <wsdl:port name="ListIncomingMaster" binding="...">
4   <soap:address location="http://www.unitn.it/BPEL4UI/ode/processes
        /FindIT/List_master"/>
5 </wsdl:port>
6 <wsdl:port name="ListIncomingSlave" binding="..">
7   <soap:address location="http://www.unitn.it/BPEL4UI/ode/processes
        /FindIT/List_slave"/>
8 </wsdl:port>
9 </wsdl:service>
10 <wsdl:service name="ListOutocoming">
11 <wsdl:port name="ListOutMaster" binding="...">
12   <soap:address location="http://www.unitn.it/BPEL4UI/
        UIEngineServer/services/FindIT/List_master"/>
13 </wsdl:port>
14 <wsdl:port name="ListOutSlave" binding="...">
15   <soap:address location="http://www.unitn.it/BPEL4UI/
        UIEngineServer/services/FindIT/List_slave"/>
16 </wsdl:port>
17 </wsdl:service>
18 ..
```

Listing 6.7 shows how the endpoints are set by the compiler, here is clear how many endpoints we needs and where they are. For the outcomes messages they points to the services deployed on the UI Engine, these links referee to the different *SOAP UI interfaces* created. Instead, the endpoints for the incomes message are provide by the BPEL Engine, in fact is the BPEL process that listen if messages are arrived or not.

6.4 Execution of the scenario

After all, is good to spend two words over the execution of the whole application and in particular for the framework part. The result page that

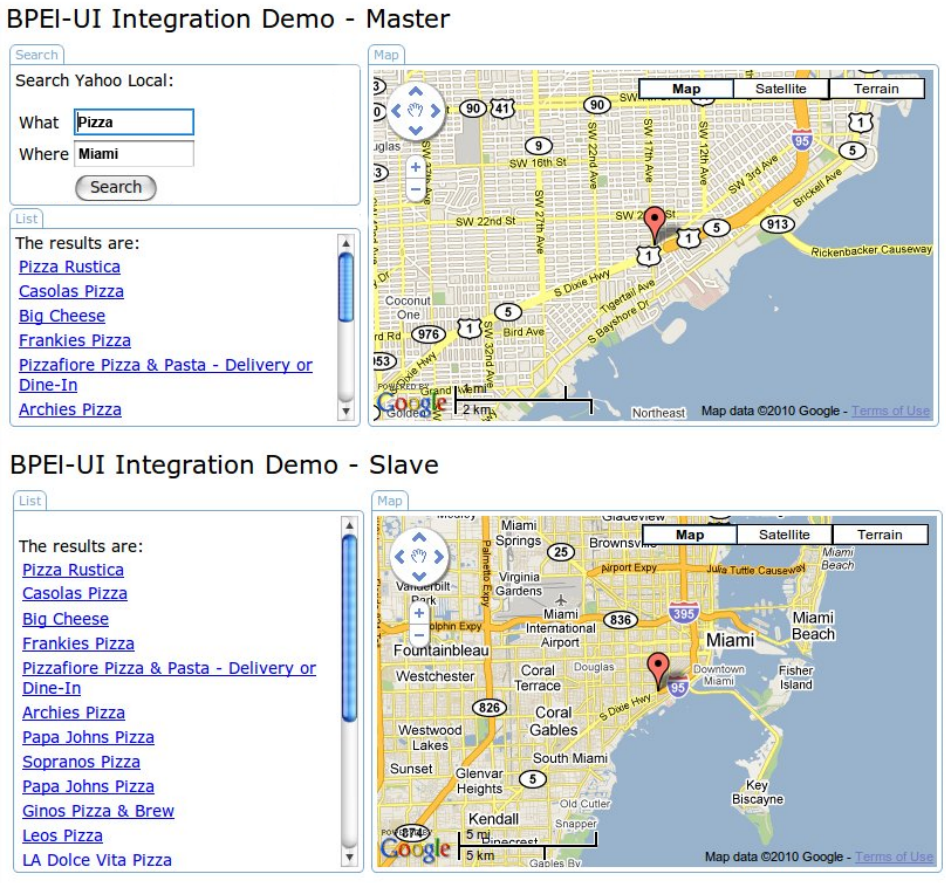

Figure 6.6: The master and the slave pages

are created from the composition, are shown as in figure 6.6, as is visible the master has the box for the search, instead, the slave has only the lists with results. Now we talks about the communication at runtime, with some practical examples done over the scenario.

6.4.1 UI2BPEL communication at runtime

This is the step where the component sends message to the process. First of all, the user has to start the application via the dedicate web application developed by us. In this way, the new page open will have an *application instance key* used to manages the communication among the different instances of the application, in this example it is "123". After that the user can start to use the page the master page, and he can start to search location, in this example he insert "pizza" as query and "Miami" as city. This request sends to the *RESTFul JSON-SOAP adapter* a request that contains the instance name that in example is "search_master", the operation name"search", and the data inserted by the user that are: "location=miami" and "query=pizza". The message passed by the component also contains the id correlation fields with the right data already assigned. When received, the service translates all these data into a well-formed SOAP message.

Listing 6.8: Example of a SOAP message created by the services

```
//Request:
instanceName = SearchForm_master
action = {"operation":"search","pars":{"what":"pizza","where":"
    miami","id_correlation":"123}}

//Result:
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
    envelope/" xmlns:loc="www.unitn.it/WSDLUI/SearchForm">
   <soapenv:Header/>
   <soapenv:Body>
      <loc:SearchElement>
         <what>pizza</what>
         <where>Miami</where>
         <id_correlation>123</id_correlation>
      </loc:SearchElement>
   </soapenv:Body>
</soapenv:Envelope>
```

Snippet 6.8 shows an example of the SOAP message generate starting from the user data, there's inside the data of the query plus the correlation id value.

6.4.2 BPEL2UI communication at runtime

Here we trade the update of the list at runtime, the message in this case is sent by the process in a soap format to the SOAP-UI interface, the message, once received is cleaned and some data are taken and removed from the original one message, substantially here there's nothing that is taken at runtime except the data of the message. From the body, the services takes the operation name *update* and, from the body of the operation, it also takes and remove the operation data and the id_correlation field (in this example is "123") , the other fields are taken and translated in JSON Object. Finally, when the service insert the event in the buffer it will insert: "List" as

name, "123" as application instance key , "List_master" as instance's name, "update" as operation's name , and a JSONObject as data. W data are simply stored in a table of a database; this table is named List, and contains all the values passed saw here (Listing 6.9 shows a practical example).

Listing 6.9: Example of a message from the BPEL process to the event buffer

```
//Request:
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/
    envelope/" xmlns:loc="www.unitn.it/WSDLUI/List">
   <soapenv:Header/>
   <soapenv:Body>
      <loc:UpdateElement>
         <ResultXML>..items in json format..</ResultXML>
         <id_correlation>123</id_correlation>
      </loc:UpdateElement>
   </soapenv:Body>
</soapenv:Envelope>

//Event inserted
name = "List"
insance-name = "List_master"
application-instance-key = "123"
operation = "update"
json-message = {"ResultXML","..items in json format.."}
```

One part of the communication is done, now there's the polling system that needs an explanation. The request is done by the UI component that performs the poll, it sends as request: as application instance's key the value 123, as parameters: operation with the value "update", as index a value depending on the last results (if the last results is the 4th, now the index will be 5 etc) as instance name "List_master". In this way, the query on the database will give the result inserted before, the instance name and the application instance id identifies the right event raised.(Listing 6.10 shows a practical example).

Listing 6.10: Example of a request for the polling and the resulting event

```
//Request:
name = "List"
application-insance-key="123"
Parameters={"opname":"update","index":"1","instance":"List_master"}

//Event read
{"ResultXML","..items in json format.."}
```

6.4.3 Multiple instances

The multiple instances, both for component and application, is probably already clear with the example saw before. Inside an application, each event contains the name of the component and also the instance's name value, so each event regards a specific component and a specific instance. In the

scenario case we have a multiple instance of the same application, we use the application instance key to discriminate among the different events inside the buffer. E.g: In the slave page, if it is opened with the key 123 the user will see all the search done by the master that uses the same key. If the master closes the page, and then it is reopened using the same key the pooling will start to re-show all the past data presented in the buffer. In this way we can navigate through the events for each user in each instance of application.

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. In this project, 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 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.

The work done for producing this thesis led to significant modifications about the management of UI components in a distributed scenario. We provided a solution that manages components using web services; this give the opportunity to exploit the power of this framework in order to construct distribute application that can be multi-browsers orchestrated. The possibility to coordinate and manage visual components among services is something new in the IT world; there is no other solution that achieves this goal. This solution can introduces some different scenarios: the data in a UI component can be inserted from different business layers or multi pages are managed by a centric point of coordination and so on. Substantially the framework, using services, permits the so-called orchestration of UI components.

7.1 Plus and minus of MarcoFlow

MarcoFlow project can be described as a new paradigm of workflow, with this solution is possible to create distributed application in a workflow fashion: the actors are involved in the process flow and they have inputs and outputs of works directly in the process description. MarcoFlow can be also thought as a standard orchestrator language with the possibility to control directly UI components and users. With this solution we provide a method to create these kinds of scenario, from the model of the process, to the final application.

MarcoFlow removes the gap existing in the orchestration of UI components, we propose a model to describe this orchestration and also a runtime framework that maintains the components orchestrated. This can be seen as a new generation of Mashup tool, more complex both in the describable logics and in its process creation. This solution is created for a highly skilled developer, with knowledge about the BPEL process language and its functionality.

The applications, which can be created with this tool, can be inserted in the class of the *event based coordinated and distributed applications, with a regard at the management of actors.* This prototype can be used to describe all the scenarios that have a complex logic, with need for multiple actors' collaboration. It can be useful for bureaucratic or complex scenarios where typically there is more than one actor in a flow of information, and where, the exchanging of messages, is frequent but slow. E.g is the scenario presented in the introduction about the home assistance process in the Province of Trento. Our system can be used to create this kind of application and then makes all more fast, the information (documents) will be immediately available at the actors without the waiting time due to the delivery of documents. For a simple use, instead, this solution can be too powerful. If we think about the FindIt single user scenario, probably, our solution is not the best choice, that scenario can be simply reproduced with a standard mash up tool in a simpler way.

In this moment, this prototype is not able to address all kinds of applications, and doesn't have a fully functionality for all the goals that it probably can solve, in example: there's no possibility to have a web services based streaming of the data inside a process. However, our solution needs some extra extensions, principally in the security field. Since now we trade actors for single pages in the process but we don't have a way to manage different policies for a specific group of users or for a single user. Also the broadcasting of messages for each instance of a component is not implemented yet, but will be useful: a component like this will permit to send with only one *invoke* messages at many users. Finally, in 3 points we can assert that Marco Flow:
Is not recommended for:

- Simple scenarios, the construction of a BPEL-UI process can be too much effort for a simply goal, there are better solutions.

- Simple standalone application, where the user is alone without collaboration with other actors, for this proposes the standard mashup works fine.

- High security and trusted process, since now the prototype has not implemented a layer that is able to manage correctly complex policy and roles. MarcoFlow is able to manages roles, that is useful for some goals but not for all.

Can be used for:

- Applications that are already thought in a workflow fashion, it can be a BPEL language or an old style workflow, where actors and documents have to collaborate in real time. Especially if the application has a user interface.

- Application where the users have to be coordinated, where the exchanges of information among users are frequent and event based.

- All the applications that are too complex to be orchestrated for a standard mashup tool, that are processes where the flows' logics is complex or a process which needs a users collaboration in multi browsers fashion.

Substantially all complex applications today involve services and humans, involve service interfaces and composition, and involve user interfaces and compositions. Yet, there are only technologies to do this for pieces. There is no technology to implement composite applications that have services and UI component, since now.

Bibliography

[1] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, May 2006. [cited at p. 17]

[2] Russell Butek. Which style of wsdl should i use? `http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/`. [cited at p. 13]

[3] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. `http://www.w3.org/TR/wsdl`. [cited at p. 12, 43]

[4] Florian Daniel, Fabio Casati, Boualem Benatallah, and Ming-Chien Shan. Hosted universal composition: Models, languages and infrastructure in mashart. In *ER '09: Proceedings of the 28th International Conference on Conceptual Modeling*, pages 428–443, Berlin, Heidelberg, 2009. Springer-Verlag. [cited at p. 38, 39]

[5] Florian Daniel, Stefano Soi, Stefano Tranquillini, Fabio Casati, Chang Heng, and Li Yan. From people to services to ui:distributed orchestration of user interfaces. 2010. in proceedings at the BPM conference 2010. [cited at p. 5]

[6] Florian Daniel, Jin Yu, Boualem Benatallah, Fabio Casati, Maristella Matera, and Regis Saint-Paul. Understanding ui integration: A survey of problems, technologies, and opportunities. *IEEE Internet Computing*, 11(3):59–66, 2007. [cited at p. 19]

[7] Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995. [cited at p. 15]

[8] Hugo Haas. Wsdl 2.0: What's new. `http://www.w3.org/2004/Talks/1117-hh-wsdl20/`. [cited at p. 12]

[9] Hugo Haas. Reconciling web services and rest services. w3c, 2005. `http://www.w3.org/2005/Talks/1115-hh-k-ecows`. [cited at p. 14]

[10] ibm.com. Ws-bpel extension for people, 2007. `http://www.ibm.com/developerworks/webservices/library/specification/ws-bpel4people/`. [cited at p. 17]

[11] Dimka Karastoyanova, Tammo van Lessen, Frank Leymann, Jörg Nitzsche, and Daniel Wutke. WS-BPEL Extension for Semantic Web

Services (BPEL4SWS), Version 1.0. Technischer Bericht Informatik 2008/03, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, April 2008. [cited at p. 17]

[12] Frank Leymann, Dieter Roller, and Satish Thatte. Goals of the bpel4ws specification. `http://xml.coverpages.org/BPEL4WS-DesignGoals.pdf`. [cited at p. 18]

[13] Raul Medina-Mora, Harry K. T. Wong, and Pablo Flores. Actionworkflow as the enterprise integration technology. *IEEE Data Eng. Bull.*, 16(2):49–52, 1993. [cited at p. 15]

[14] Jörg Nitzsche, Tammo van Lessen, Dimka Karastoyanova, and Leymann Frank. *Business Process Management*. springerlink, 2007. `http://www.springerlink.com/content/l255455h18n8nnp6/`. [cited at p. 17]

[15] oracle.com. Manipulating xml data in bpel. `http://download.oracle.com/docs/cd/B31017-01/integrate.1013/b28981/manipdoc.htm`. [cited at p. 47]

[16] Cesare Pautasso. Restful web service composition with bpel for rest. *Data and Knowledge Engineering*, 68:851–866, 2009. [cited at p. 17]

[17] Tammo van Lessen, Leymann Frank, Mietzner Ralph, Jörg Nitzsche, and Daniel Schleicher. A management framework for ws-bpel. In *ECOWS '08: Proceedings of the 2008 Sixth European Conference on Web Services*, pages 187–196, Washington, DC, USA, 2008. IEEE Computer Society. [cited at p. 18]

[18] w3c.org. Web services glossary. `http://www.w3.org/TR/ws-gloss/`. [cited at p. 11]

[19] w3c.org. Web services choreography description language version 1.0, 2007. `http://www.w3.org/TR/ws-cdl-10/`. [cited at p. 17]

[20] Jin Yu, Boualem Benatallah, Regis Saint-Paul, Fabio Casati, Florian Daniel, and Maristella Matera. A framework for rapid integration of presentation components. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 923–932, New York, NY, USA, 2007. ACM. [cited at p. 38]

Ringraziamenti

Finalmente una parte interessante: i ringraziamenti (sperando di non dimenticare nessuno)

Un grazie dovuto e voluto a chi ha lavorato con me su questo progetto, in rigoroso ordine alfabetico: Fabio, Florian, Michele, Stefano e Valeria.

Prima degli amici vorrei ringraziare i miei genitori e in particolare mia *mamma* e mio *papá* (cit. Altobelli, una frase che ho sempre voluto utilizzare anch'io, e quale maniera migliore di questa?), mia *sorella* e anche Damiano per essere sempre presenti alla fase ingrasso di mamma rosi e anche per tutto il resto. Grazie anche allo *zio*, alla *nonna* e a tutto il parentado.

Ora gli amici vari: quelli di una vita, quelli di una sera, quelli della biblio, quelli dell'uni, quelli del calcio, quelli! I migliori probabilmente, tutti qui (sperando di non averne dimenticati). In rigoroso ordine casuale (alcuni sono raggruppati *ndr*) generato con un apposito programma (non ho mica studiato informatica per nulla): Chess, Martina, Rico, Öltien straz, Benetti l'olfo, Musso detto Andrea, Marchi il conte, poser Parisi, panchina la domenica Festi, l'Erko, l'asd SaccoSanGiorgio, Omino, Lorentz, Annina x2, Sega Ga boulder, Mario Ulisse, Seba Conte oh conte, l'Alby e l'Eli, la massa critica, Samantharra, il Giast, Colo senior e Colo junior, la Bionda, Tommy 33 Trentini, Tia e la Vale, il Lino, Milhouse, il Biondo, la Maga Ila, tutti quelli che mi sono dimenticato e tutti quelli che non ho ancora incontrato.
Vi voglio bene.

Stefano