SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0; **0**:1–0 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

A Layered Architecture for Flexible Web Service Invocation

Valeria De Antonellis[1], Michele Melchiori[1], Luca De Santis[2], Massimo Mecella[2], Enrico Mussi[3]*, Barbara Pernici[3], Pierluigi Plebani[3]

[1] *Università di Brescia, Via Brianze 38 - 25123 Brescia, Italy*
[2] *Università di Roma "La Sapienza", Via Salaria 113 - 00198 Roma, Italy*
[3] *Politecnico di Milano, Piazza Leonardo da Vinci 32 - 20133 Milano, Italy*

SUMMARY

The use of Web services composition is emerging as an interesting approach to integrate business applications and create intra-organizational business processes.

Single Web services are composed to create a complex Web service which will realize the business logic of the process. Once the process is created, it will be executed by an orchestration engine which will invoke individual Web services in the correct order. Sometimes it can occur that some Web services composing the workflow are no longer available during the run-time phase, blocking the process execution.

In this paper we describe an architecture which allows the orchestration of business processes in a flexible way. With our approach, Web services composing the process can be automatically substituted with other compatible Web services during process execution. A methodology to evaluate Web services compatibility in order to select substitutable Web services is defined.

KEY WORDS: Web services, Orchestration, Dynamic Process Evolution, Substitution

1. Introduction

Web services are emerging as a key technology for the integration of applications within and across enterprise boundaries [9].

A Web service can be defined as 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 WSDL). Other systems interact with the Web service in a

*Correspondence to: Enrico Mussi, *Dipartimento di Elettronica e Informazione*, *Politecnico di Milano*, Piazza Leonardo da Vinci, 32 - 20133 Milano, Italy. E-mail: mussi@elet.polimi.it

manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards [1].

With Web services it is possible to wrap and reuse legacy applications in inter-organizational processes and, in this way, to deploy services that can be easily used to create cooperative processes, i.e., a set of collaborating activities in order to reach a common goal. According to that, Web services may be deployed by different providers and the overall process can be distributed across multiple organizations.

Many efforts have been made by the Web service community to define standards and protocols in order to enable system integration. Among others, emergent technologies such as BPEL4WS [15], WSCI [3] and BPML [4] enable Web services orchestration and choreography. With these technologies, indeed, it is possible to define abstract business processes and then execute their instances [22]. Unfortunately, none of these approaches allows Web services substitution at run-time. The more interesting case is represented by BPEL4WS [†] where the process designer can define an *abstract process model* composed by the typical workflow structures and where the activities are specified according to the WSDL of the wished Web services. Once this model is created for each specified Web service, real deployed Web services must be selected to build an *executable process*.

This approach, however, fails with respect to the provisioning of the basis for a real abstract process specification for two main reasons. First of all, given a Web service description required by the *abstract process model* is almost impossible to find more than one available Web service with exactly the same syntax, indeed, usually the process designer defines the *abstract process model* knowing a priori the set of Web services which will be used in the *executable process*. Secondarily, at run time, the *executable process* is fixed and there is no way to modify any invoked Web service.

Consequently, it is necessary to exploit the differences between what is desired and what is real using *abstract* service definitions and then create an architecture to dynamically invoke the *concrete* deployed services in place of the *abstract* ones. The idea is that each abstract service can be substituted with a concrete compatible service and then, during the process execution phase, the architecture can choose which concrete service to invoke.

In previous works [11, 14, 19, 20] we have developed theories and methodologies for dynamic Web services substitution and their orchestration; the aim of this paper is to present the definition, design and implementation of the architecture for the dynamic management of cooperative processes, that we have realized in the context of the Italian research project VISPO *(*V*irtual-district* I*nternet-based* S*ervice* P*latf*O*rm)*. In such a project, different organizations of a district cooperate on the basis of a cooperative process, were all activities are implemented with the Web service technology. Our architecture (and the implemented platform) *(i)* supports the definition of abstract processes and executes process instances, and *(ii)* allows automatic run-time Web service substitution. With our approach there is no a priori defined limit on the number of invocable Web services.

[†]BPEL4WS is now under control of OASIS consortium and renamed as WS-BPEL. Even if this moving has introduced some modifications to the specification, the conceptual model remains the same and such changes do not affect our work.

The prominent aspects of our approach are:

- *Abstract service* definition. The introduction of the abstract service concept allows us to categorize Web services and gives us the ability to describe the cooperative process in an abstract way.
- *Concrete service* definition. Concrete services represent deployed Web services. They are organized according to the functionalities described by abstract services and realize the activities of the cooperative process.
- *Service compatibility* definition. At design-time, a cooperative process is composed by abstract services. During the execution, abstract services are substituted by concrete services which can be invoked. Substitution is possible only between compatible services and so an evaluation methodology is necessary.

The rest of the paper is organized as follows. Section 2 introduces the case study that was used to validate our architecture. Section 3 describes how service compatibility is evaluated and presents our architecture for dynamic service substitution. Section 4 describes how a cooperative process is executed in our approach. Section 5 analyzes performance aspect of the deployed system. Finally, Section 6 draws some concluding remarks.

2. Case Study

The case study adopted in this paper to describe a sample application of our methodology is the execution of a cooperative process in a virtual district, that is a productive district where involved companies use ICT (Information and Communication Technology) to cooperate for business purposes. In particular, we consider a cooperative process called *"Advanced Purchasing"* .

The execution environment is an ASP (Application Service Provider) platform that the virtual district uses for managing its business. We suppose that similar ASP platforms are used in different districts. The "Advanced Purchasing" process is composed of three services that are coordinated in order to accomplish a common goal: processing district purchase orders in an automated and sophisticated way. Figure 1 shows the workflow that defines the cooperative process used for testing our methodology. The "Advanced Purchasing" process can be viewed as a service itself.

Every single service composing the process performs specific operations, whereas the district ASP is in charge of coordinating communication and synchronization between services, thus acting as a service orchestrator.

The three services used for composing the cooperative process are described in the following. These services provide all the necessary operations to implement advanced purchasing functionalities and are invoked by the process during its execution.

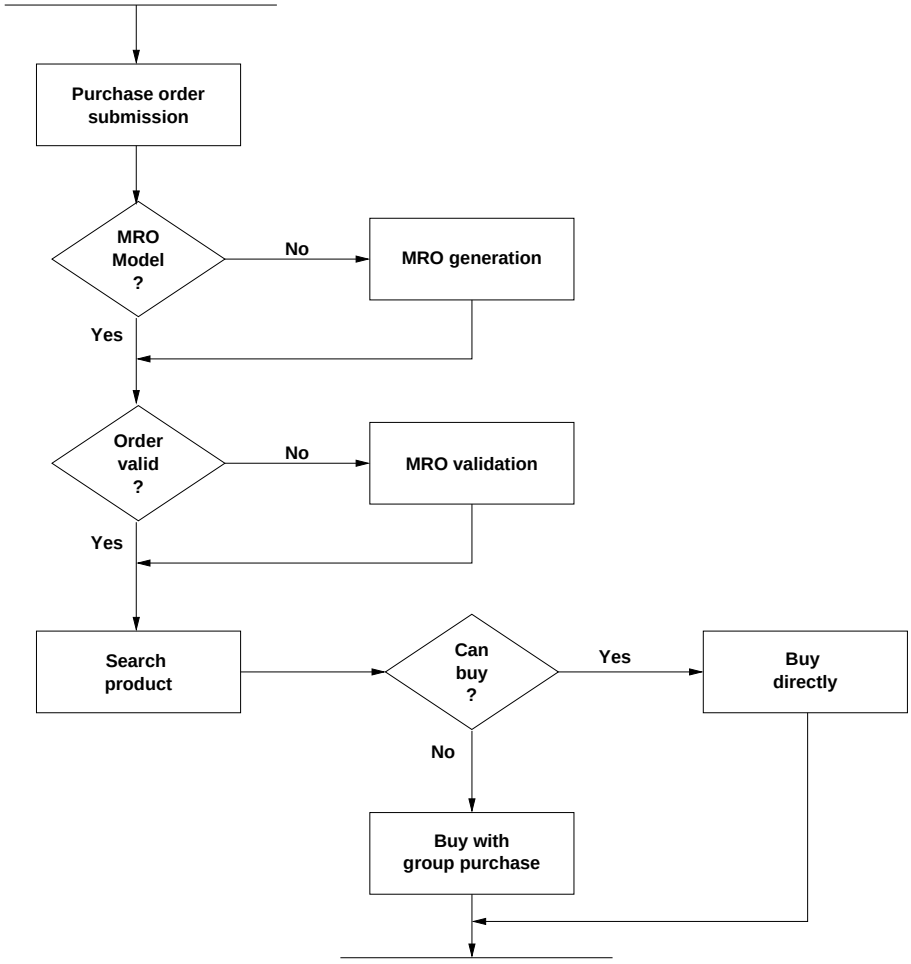

Figure 1. Advanced Purchase Process Workflow

- **MRO Service:** it allows generating a consumption model for a company[‡]. The model is created by analyzing previous purchase documents (e.g., invoices), and collects a normalized description of ordered products. Given a purchase order and a consumption model, the MRO Service can state if the order of a product belongs to the model and, in

[‡]In the specific economic slang, the MRO refers to all consumable goods used by a company, e.g., pens, papers, gloves, hardware, etc.

this case, return the normalized description of the product that is needed by the other involved services. This service provides operations to perform the *MRO Generation* and *MRO validation* activities.

- **e-Selling Service:** it acts like a virtual catalog. Companies use the e-Selling Service to publish their catalogs or to search and buy products from other companies of the virtual district. Buyers can access the catalog, search for products and buy them directly through this service. Operations provided by this module allow performing the *Search product* and *Buy directly* activities.
- **Group Purchases Service:** it allows grouping purchase orders of various enterprises in order to simplify and optimize purchase procedures. Moreover, by using the Group Purchases Service, enterprises increase their contractual power with respect to their suppliers and obtain better discounts. With this service is possible to perform the *Buy with group purchase* activity.

Although the *e-Selling* and *Group Purchases* services are strictly related to a particular ASP platform, the *MRO* service can be considered more general. We suppose that the "Advanced Purchasing" process, executed into a virtual district $\mathcal{A}$ can be performed even with a *MRO* service taken from another district $\mathcal{B}$.

As an example, to be used throughout the paper, in a specific "Advanced Purchasing" process instance, deployed as an orchestration of Web Services, the following interactions can take place:

1. the cooperative process starts with a purchase request;
2. the orchestrator invokes the *MRO* Web service for generating the correct consumption model;
3. the consumption model is generated and returned to the orchestrator;
4. the order is valid with respect to the consumption model and the orchestrator asks to the *e-Selling* Web service if the purchase request can be satisfied;
5. the *e-Selling* Web service notifies that products are not available;
6. the orchestrator invokes the *Group Purchases* Web Service;
7. the *Group Purchases* Web service accepts the purchase order, executes it and returns the response to the orchestrator, thus ending the process.

Figure 2 shows messages exchanged between the orchestrator and Web services; in this case the orchestrator is in charge of executing the cooperative process specification by sending messages to proper Web services and managing responses correctly.

We also suppose that, during a process instance execution, the *MRO* Web service becomes no more available; when this occurs, the orchestrator must substitute the original Web service with a new compatible one in order to carry on the process. The new Web service might have a different interface but should implement the same functionalities, thus being an effective possible substitute. Figure 3 depicts the substitution scenario, in which the orchestrator retrieves the compatible Web service *MRO Utilities* taken from the district $\mathcal{B}$ and invokes it in order to proceed with the process execution.

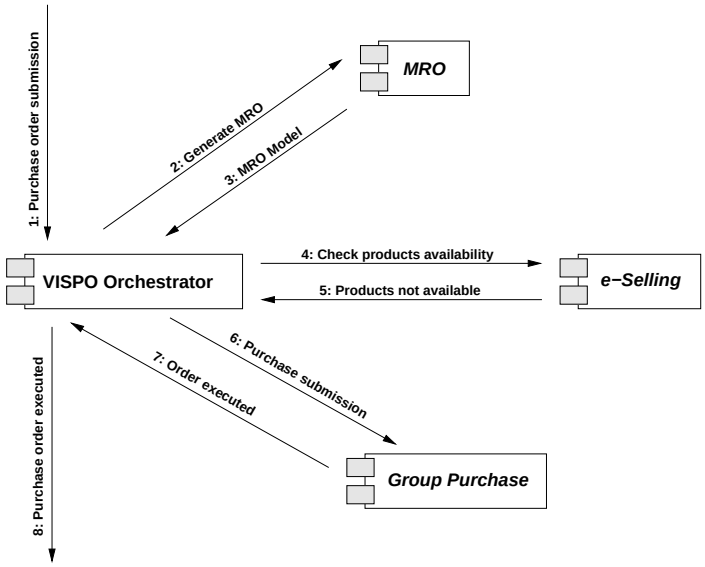

Figure 2. Advanced Purchase Process Service Interactions

3. Design Issues and Architecture

The main purpose of the proposed architecture, referred to as VISPO *(*V*irtual-district* I*nternet-based* S*ervice* P*latf*O*rm)*, is to implement a computational model based on the Service Oriented Computing [21] paradigm to execute a *cooperative process* with flexible and dynamic Web service invocation.

A cooperative process can be defined as a set of collaborating services in order to reach a common goal. Each service executes a particular task and it is invoked by the process execution engine. Invoking Web services in a flexible way means the ability for the execution engine to replace Web services during the process execution.

In this section we describe our approach to evaluate services compatibility and present the architecture of the VISPO system which implements the compatibility evaluation and the process execution with dynamic substitution of Web services.

There are three kind of users who interact with the VISPO system:

- Users, who interact with our platform for executing cooperative processes.
- Domain experts, who supervise the VISPO system for controlling and managing the performed activities.
- Service providers, who create abstract and concrete services and publish them into the VISPO system registry.

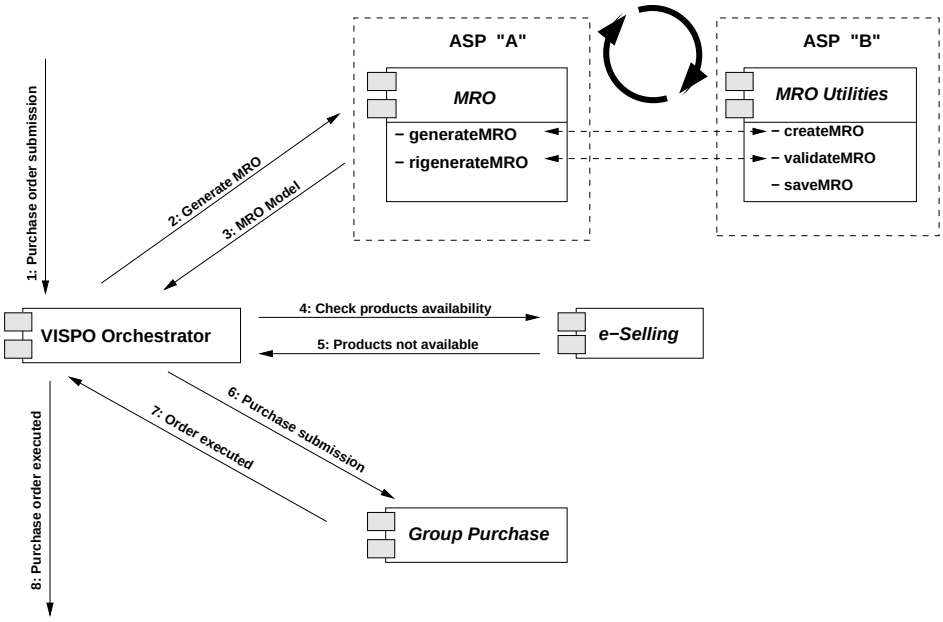

Figure 3. Advanced Purchase Process Service Substitution

3.1. Flexible Service Invocation

Executing an abstract cooperative process in a flexible way means to have the ability to substitute one or more Web services used in the process when this is required or appropriate; for instance, possible cases in which this may be useful are:

- an updated version of an existing service is made available;
- a new service that provides better features and is compatible with a service already used in the process is published;
- a service used in the process becomes no more available;
- at each execution of the cooperative process the more suitable service to implement an activity is selected from a set of equivalent services on the basis of criteria like the current cost, the current availability, etc.

Obviously a substitution should not involve any possible service but only services that have similar functionalities. Therefore, in the following we introduce the concept of *compatibility class*, as a set of services that are compatible with respect to the execution of a process activity.

For example, in Figure 3 the *MRO* service is substituted by the *MRO Utilities* service that has similar functionalities. This means that both services belong to the same compatibility class.

Moreover, we define an approach to evaluate the service compatibility and a mechanism for substituting Web services during their invocation.

Our approach is based on the Service Oriented Architecture [23] where: *(i)* services are registered by providers into a registry, *(ii)* a service requester accesses to the registry for retrieving a service and gets the reference by means of which the service can be invoked.

Unlike current approaches, our service registry implementation, called VISPO *Registry* and fully described in the next section, provides a specific categorization which organizes the registered Web service in compatibility classes. The service provider associates a service to the compatibility class he thinks the service is able to satisfy. Such requirements are stated when the class is created and have to be satisfied by all class members. In order to help the service provider to properly register the service, a set of semi-automatic mechanisms studied in [14], and based on the semantic of the Web services, is provided. In this way when the service provider is registering a Web service, these mechanisms, on the basis of the structure of the WSDL, recognize which are the set of compatibility classes closer to the Web service. During the registration, compliance of the registered service to its compatibility classes is also evaluated by the VISPO system and mapping information for semi-automatically building wrappers to adapt services to the process is generated. In this way, the registry, for each compatibility class defines a rank of services according to the satisfaction of the requirements expressed by the compatibility class. An higher position in the rank means higher compatibility and less work to build wrapper. On the contrary, a lower position in the rank means lower compatibility and a more complex wrapper.

Compatibility classes are the basis of our substitutability approach and allows referring to a set of services, i.e., the concrete services, through a sort of representative of them, i.e., the abstract service. In this way the cooperative processes may be defined as a composition of abstract or concrete services where their invocation is performed in a flexible way. In case concrete services are indicated, the process can be immediately enacted since all the information requested by the orchestrator (i.e., messages, endpoint reference, etc.) is already known. In the second case a selection phase is required in order to identify which is the best concrete service that can perform the function specified by the abstract service. In any case, during the process execution, the running service can be substituted by the services belonging to the same compatibility class the concrete service belongs (or the abstract service represents) using the compatibility rank previously defined.

The following section describes how the service compatibility can be evaluated and subsequently substituted at run-time.

3.1.1. Service Description

The compatibility evaluation takes into account descriptions that are expressed by means of a suitable service description language.

Currently, WSDL is the de-facto standard for describing Web service functionalities. A WSDL specification represents the structure of a Web service in terms of provided operations, and for each of them the data requested and returned. In particular, a WSDL `portType` defines a collection of abstract operations supported by the service, separating them from their actual implementation. In our approach, we suppose that services are described as WSDL 1.1

Table I. WSDL entities and corresponding Descriptor concepts

WSDL entity	Descriptor concept
PortType name	Descriptor name
Operation name	Operation name
Input message part name	Input entity name
Output message part name	Output entity name

[16] documents and that each Web Service has only one associated `portType` describing the operations that the service implements. This implies that a service presenting several interfaces (i.e., `portTypes`) needs to be represented as a set of Web services.

WSDL, according to [6], enables Web service publishers to separate the abstract definition of service functionalities from the specific details of implementation, such as service location and service access protocols. We adopt the definition of *abstract service* for definitions of services restricted to the only service interface, while fully described services are called *concrete services*.

This distinction is very useful in the definition phase of the cooperative process because it allows describing the process from an abstract point of view in terms of abstract services and it lets the VISPO runtime environment decide which concrete services to invoke.

Moreover, our definition of abstract services implies that they are composed by the `types`, `message` and `portType` WSDL 1.1 elements. Such a definition is in accordance with the one defined in the BPEL4WS and BPML specifications and allows us to easily integrate abstract services into these orchestration languages.

Candidates to be invoked are the concrete services belonging to the same *compatibility class* of the abstract service. In our system, an abstract service definition introduces an associated compatibility class whose members will be concrete services. These concrete services may either share the abstract service interface associated to the compatibility class and therefore they implements the abstract service or have a different interface, to be used through a wrapper component.

Therefore, the creation of a compatibility class happens when an abstract service is registered; a concrete service is registered associating it to one or more compatibility classes. The next subsection introduces the evaluation of the compatibility degree among the members of a compatibility class and the representative abstract service.

3.1.2. Service Compatibility

In VISPO, the compatibility evaluation between services is performed on the basis of semantic information related to service interfaces. For this purpose, each Web service is represented in terms of a descriptor extracted from its WSDL specification (see Table I). Descriptors give a summary abstract view of services in terms of deployed operations and exchanged information.

Approaches based on the use of descriptors are widely studied in the field of reusable software components [13] for discovering components in a library that match with given requirements. A descriptor is formally defined by a name of service and a set of triplets:

$$< \text{operation}(OP), \text{input entities}(IN), \text{output entities}(OUT) >$$

where:

- OP is the set of operations a service can perform;
- IN is the set of the input information entities;
- OUT is the set of the output information entities.

Figure 4 shows an example of a WSDL `portType` definition, while Figure 5 shows the corresponding descriptor.

Descriptors are analyzed to compute similarity coefficients (formally described in 3.2.2) so that the higher the similarity coefficient value the higher the similarity of the involved services in terms of operations they provide and data they exchange with the invoking process.

Note that a high similarity value between service descriptors does not guarantee the possibility of a full automatic substitutability between the corresponding services. In fact, there could be syntactic differences among involved interfaces, that is different operation and parameters names, parameter number and parameter positions (as shown in Figure 3, where the *MRO Utilities* service and the *MRO* service differ in the number and name of operations). Therefore, besides the similarity evaluation of descriptors, is necessary to create information, here called *mapping information*, used to take into account the differences between an abstract service interface and a concrete service interface. Figure 6 reports an example of mapping information.

It is worth noting that both service compatibility evaluation and mapping information generation have a 1:1 approach. Given two services, each part of the first service is compared with each part of the second service and the relative mapping information is generated; therefore each operation is compared with a single operation and not with the composition of two or more other operations.

3.1.3. Service Registration and Retrieval

The publication of a Web service is obtained by registering its WSDL specification into the VISPO Registry and specifying its compatibility classes. According to the type of service the provider is going to publish, different steps should be followed. In particular, the publication of an abstract service implies the definition of a new associated compatibility class. On the contrary, service providers who want to publish a concrete service must first define their Web service interface and then register it into one or more existing compatibility classes.

Besides the publication mechanisms typical of the common registries, the VISPO Registry at the same time of the registration of a service generates the correspondent descriptor and performs the similarity evaluation with respect to the abstract services corresponding to the compatibility classes in wich the service is inserted. In particular, the descriptor generation phase is transparent to the publisher in order to maintain the compatibility evaluation logic

```
<types>
...
<element name="Category">
 <complexType>
  <sequence>
        <element name="categoryName" type="xsd:string"/>
        <element name="categoryID" type="xsd:decimal"/>
   <sequence>
 <complexType>
...
</types>
...
<wsdl:message name="generateMroRequest">
        <wsdl:part name="businessName" type="xsd:string"/>
        <wsdl:part name="fileInput" type="xsd:string"/>
</wsdl:message>

<wsdl:message name="generateMroResponse
        <wsdl:part name="mroID" type="xsd:string"/>
        <wsdl:part name="businessName" type="xsd:string"/>
        <wsdl:part name="productCategory" type=" Category "/>
</wsdl:message>

<wsdl:portType name="Mro">
        <wsdl:operation name="generateMro" parameterOrder="businessName fileInput">
        <wsdl:input name="generateMroRequest" message="impl:generateMroRequest"/>
        <wsdl:output name="mroGenerateResponse" message="impl:mroGenerateResponse"/>
</wsdl:operation>
...
</wsdl:portType>
```

Figure 4. WSDL `portType` example

internal to the system. For each compatibility class in which the concrete service is inserted, similarity values are computed, and mapping information is generated.

3.1.4. Wrappers and Service Invocation

Once compatibility is evaluated and mapping information generated, concrete services can be invoked. Our invocation technology allows mapping each abstract service interface to a concrete service implementation and lets the cooperative process runtime environment act as if it was invoking abstract services.

When an activity of a running cooperative process has to be executed and the corresponding abstract service is invoked, two different cases are possible:

- the interface of the concrete service is the same of the abstract one, then the system can locate and invoke the concrete service directly;
- the abstract and concrete interfaces are different, therefore a wrapper is needed to map abstract operations/parameters to concrete operations/parameters.

```
<descriptor>
 <name>Mro--uuid:19758A80-FE3E-11D7-B93D-9B8ABDDEB39C</name>
 <wsdlKey>uuid:19758A80-FE3E-11D7-B93D-9B8ABDDEB39C</wsdlKey>
 <compatibilityClass>MroServices</compatibilityClass>
<operations>
 <operation>
  <operationName>generateMro</operationName>
   <inputParameters>
    <inputParameter>
      <inputParameterName>businessName</inputParameterName>
      <inputParameterType>string</inputParameterType>
    </inputParameter>
    <inputParameter>
      <inputParameterName>fileInput</inputParameterName>
      <inputParameterType>string</inputParameterType>
    </inputParameter>
   </inputParameters>
   <outputParameters>
    <outputParameter>
      <outputParameterName>mroID</outputParameterName>
      <outputParameterType>string</outputParameterType>
    </outputParameter>
    <outputParameter>
     <outputParameterName>businessName</inputParameterName>
     <outputParameterType>string</inputParameterType>
    </outputParameter>
    <outputParameter>
     <outputParameterName>productCategory</inputParameterName>
     <outputParameterType>Category</inputParameterType>
    </outputParameter>
   </outputParameters>
 </operation>
</operations>
</descriptor>
```

Figure 5. Descriptor Example

An interesting case is represented by a concrete service which presents an affinity value equals to 1. In this case, even if the concrete service fully satisfies the requirements of the abstract service, a wrapper could be necessary. In fact, the affinity value does not pay attention to the order of the operation parameters and the maximum affinity value can be given to a concrete service even if the concrete operation signatures do not completely match the abstract operation signatures; in this cases, a wrapper solving these discrepancies is required.

Domain experts have the responsibility of creating wrappers on the basis of the mapping information. A facility in the VISPO system allows creating a wrapper skeleton that the domain experts can check/modify to obtain the desired adaptation result (see Section 3.3).

```
<mappingInformation>
 ...
 <operationBindings>
  <operationBinding>
   <abstractOperation>
    <name>generateMro</name>
   </abstractOperation>
   <concreteOperation>
    <name>create</name>
   </concreteOperation>
   <inputParameterBindings>
    ...
   </inputParameterBindings>
   <outputParameterBindings>
    ...
   </outputParameterBindings>
   <affinityValue>1.0</affinityValue>
  </operationBinding>
 </operationBindings>
 ...
</mappingInformation>
```

Figure 6. Example of mapping information

3.2. Architecture

On the basis of the previous ideas, we introduce the VISPO architecture that supports flexible Web services invocation.

Figure 7 shows the system architecture, which consists of four main components:

- the VISPO *Registry* to register and search Web services and their descriptions;
- the *Compatible Service Provider (CSP)* to retrieve all *concrete* services belonging to a *compatibility class* and define the *mapping information* needed for service substitution;
- the *Compatibility Module* to perform descriptor similarity evaluation and create mapping information;
- the *Invocator* to perform the flexible invocation of concrete services;
- the *Orchestration Engine* to orchestrate services invocation.

In summary, the execution of the activities in the process is based on the Invocator functionalities: it receives an abstract service definition as input and then uses the Compatible Service Provider to find all the compatible services and retrieve the related mapping information. On the basis of this information, a concrete service is actually selected and invoked.

In the following, we present the details of the VISPO *Registry*, the *Compatible Service Provider* and the *Invocator*, while the *Orchestration Engine* is described in Section 4.

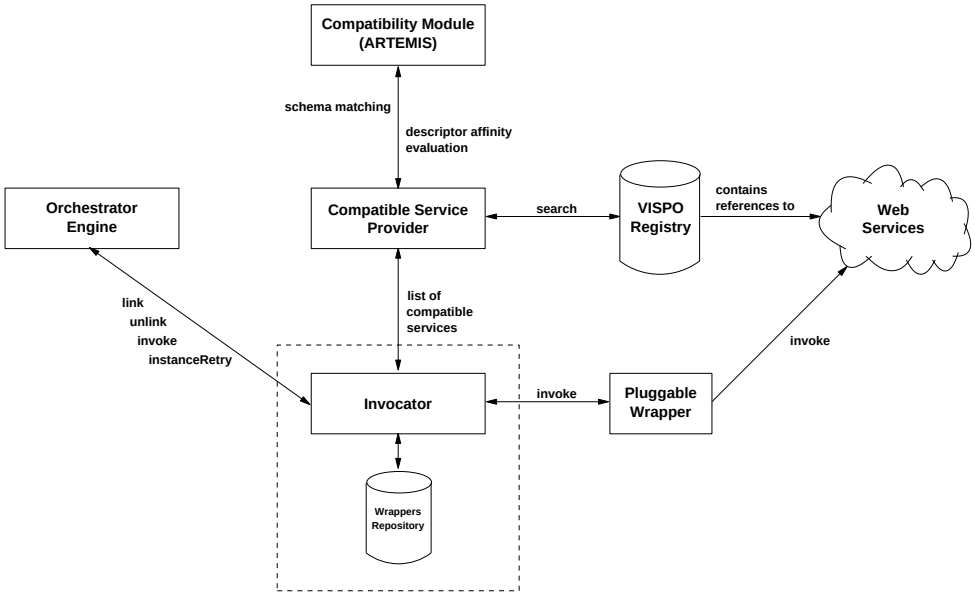

Figure 7. Global Architecture

3.2.1. VISPO *Registry*

The VISPO Registry can be used to publish and retrieve Web service specifications, group services into compatibility classes, manage service descriptors.

As Figure 8 shows, the VISPO Registry is composed of a database, called *descriptor base*, and a UDDI Registry V2 [25]. Both parts are accessed through the VISPO *Registry API*, which extends the methods of UDDI API V2 [26] in order to support the management of descriptors and maintain the compatibility with the UDDI V2 API specification. In this way, the registry can still be used by a UDDI V2 standard client for publishing and retrieving services information, while only extended clients can access descriptors using registry specific methods.

In our implementation we decided to use a UDDI V2 registry because the UDDI V3 [27] registry specification was still under development and client libraries for coding the VISPO *Registry API* were not available. However, our choice does not prevent from using a UDDI Registry V3 for realizing the UDDI component of the VISPO Registry.

UDDI V3 specification explains how to build a UDDI V3 registry with a multiple version support, which allows using UDDI V2 clients to access UDDI V3 registries. Besides giving guidelines on how to implement a registry, UDDI V3 specification also defines some constraints on data types composition that must be followed by UDDI V2 clients in order to interact with UDDI V3 registries. These constraints are satisfied by our VISPO Registry API.

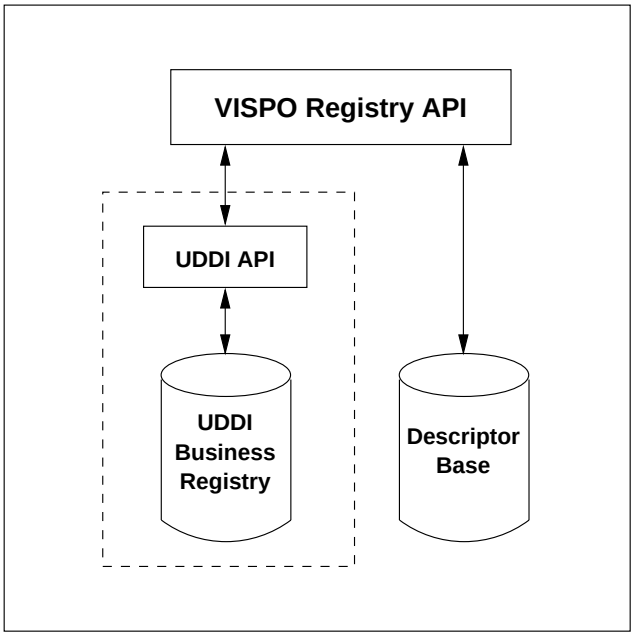

Figure 8. VISPO Registry Architecture

Service publishing is one of the most important operations and it is slightly different for abstract and concrete services. However, both kind of services are categorized using the categorization features provided by UDDI V2 registries. Using the `categoryBag` and its subelement `keyedReference` in a proper way [5], it is possible to define a *general keywords taxonomy* and categorize the published services.

The structure of the `keyedReference` is composed by three elements:

- a *tModelKey* element, which specifies `tModel` that defines the taxonomy used for the categorization. We use the *UUID:A035A07C-F362-44DD-8F95-E2B134BF43B4* value, which implies the use of a *general keywords taxonomy*.
- a *keyName* element, which specifies the name of the general keyword taxonomy that will be used for services categorization. We use the *vispo-category:types* value that states the usage of the VISPO taxonomy.
- a *keyValue* element, which specifies the value of the category. This value represents the name of the compatibility class in which the published service is inserted.

The same approach can be followed even with an ebXML Registry [12], one of the most important alternative to UDDI. In fact, ebXML Registry Model includes the possibility, through the so called *classification scheme*, to introduce a specific way to organize the information stored in such a registry.

In the following, guidelines for abstract and concrete service publication are described:

Abstract service publication can be done by saving into the VISPO Registry a `tModel` which refers to the WSDL abstract description of the service. The `tModel` is characterized by a `categoryBag` with a `keyedReference` in which the `keyValue` element contains the name of the compatibility class defined by the published abstract service.

Once the abstract service is published, the descriptor is automatically created while the VISPO Registry is in charge of checking out that every new abstract service defines a new compatibility class. It is not allowed having more than one abstract service for a compatibility class. The registry analyzes the published contents and, if inconsistencies are detected, notifies the abstract service publisher.

Concrete service publication requires to register a `tModel`, which refers to the complete WSDL description including the binding information and specifies the compatibility classes in which the concrete service is inserted. The `tModel` is characterized by a `categoryBag` with a series of `keyedReference`, each of them defining a compatibility class to which the concrete service belongs. The value of the compatibility class is inserted into the *keyValue* element and the VISPO Registry is in charge of checking that every defined *keyValue* refers to an existing compatibility class. If that don't happen, the registry notifies the concrete service publisher.

When a concrete services is published, descriptor and mapping information are generated. While the descriptor generation is performed by the registry itself, the generation of mapping information is performed by the Compatible Service Provider. Given a published concrete service, the VISPO Registry passes it to the Compatible Service Provider, which is in charge of evaluating its compatibility with respect to the abstract services of the compatibility classes in which it was inserted and generating the mapping information. The Compatible Service Provider is deeply described in the next section. However, in order to avoid wrong publications and create homogeneous compatibility classes, the domain expert must check the submitted `keyValue` values and the generated information for verifying that the concrete service is really suitable to be included in the specified compatibility classes.

3.2.2. Compatible Service Provider

Given a set of services, the Compatible Service Provider (CSP) is able to evaluate their compatibility with respect to a reference service and generate the correspondent mapping information. Once the mapping information has been generated, it is saved into a dedicated registry and can be accessed by the CSP for retrieving useful information about service compatibility.

In the VISPO system, the CSP is used during the publication and invocation phases. In the first phase, it is used by the VISPO Registry for evaluating the affinity of the published concrete service with respect to the abstract services that define the compatibility classes in which the service is published. In the latter, the Invocator uses the CSP for retrieving the compatibility

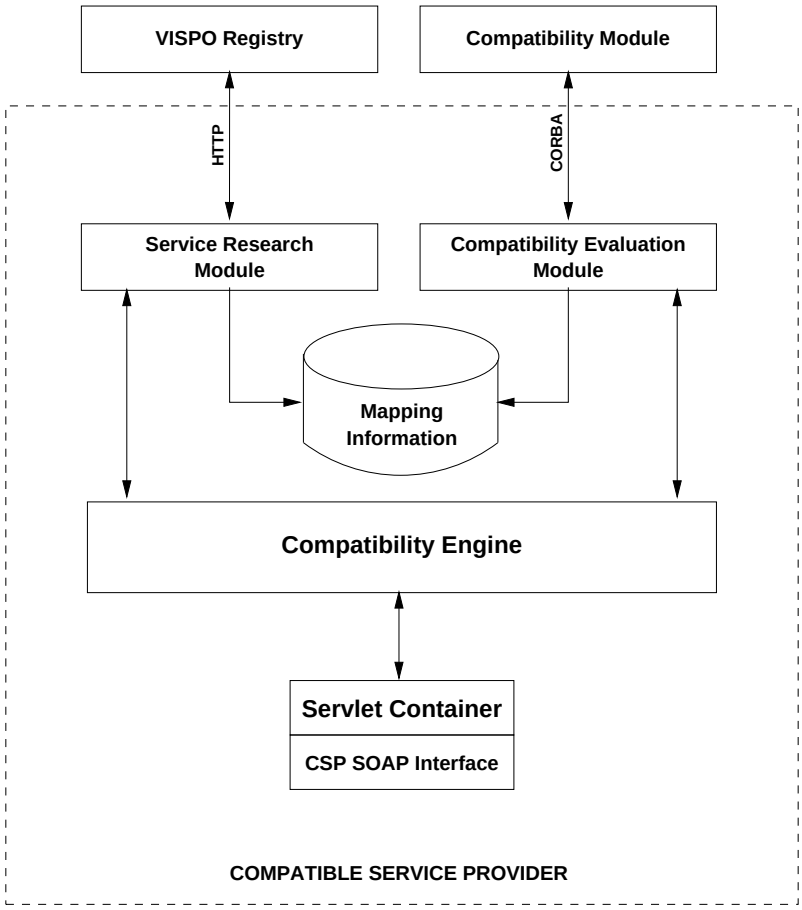

Figure 9. Compatible Service Provider Architecture

values of concrete services belonging to the same compatibility class in order to rank and select which concrete service to invoke.

The architecture of the CSP is depicted in Figure 9 and is composed of four main components:

- *Service Research Module* – The main function of the Service Research Module is to serve as a proxy for the Compatibility Engine toward the VISPO Registry. This module provides various methods for browsing the registry and retrieving service information in a useful way for the Compatibility Engine. The goal of this module is to use the simple *inquiry*

API of the VISPO Registry in order to create more complex operations for retrieving all the needed service information in only one step. With this module is possible to:

- given a service key, retrieve its WSDL and its descriptor;
- given a compatibility class, retrieve all the WSDL documents of services belonging to it;
- given a compatibility class, retrieve all the descriptors of services belonging to it.

The communication with the registry, which can reside on a different host, is done via HTTP and all exchanged information is in XML format.

- *Compatibility Evaluation Module* – The Compatibility Evaluation Module is responsible for supporting the Compatibility Engine during the service compatibility evaluation. Its goal is to provide a set of optimized operations to facilitate tasks like affinity evaluation and service filtering. Through these operations, which are built on the top of the *semantic descriptor evaluation* and *schema matching* functionalities provided by the ARTEMIS [8, 7] tool, the Compatibility Engine evaluates compatibility between services and generates the mapping information.
  With the Compatibility Evaluation Module it is possible to evaluate:
  - "similarity" between abstract and concrete descriptors;
  - "similarity" between abstract and concrete operations;
  - "affinity" between abstract and concrete input parameters;
  - "affinity" between abstract and concrete output parameters;
  - "compatibility" between abstract and concrete data types.
- *CSP SOAP Interface* – As shown in Figure 9, the Compatible Service Provider can be accessed through a SOAP interface which allows using the CSP as a Web service. This interface is built using a servlet container.
- *Compatibility Engine* – The Compatibility Engine is the core of the Compatible Service Provider. It contains the logic for searching concrete services and evaluating their affinity with respect to the abstract service belonging to the same compatibility class. The result of this process is a list of compatible services with the related mapping information. The evaluation process is composed by seven steps:
  1. *Service research*: Once received the `tModel key` and the compatibility class of an abstract service as input, the compatibility engine retrieves from the VISPO Registry all the descriptors and WSDL descriptions of the concrete services belonging to the same compatibility class. This operation is done using the research module.
  2. *Descriptors matching*: Using the Compatibility Evaluation module the abstract service descriptor is matched against all the concrete services descriptors. For each matching a similarity coefficient *GSim* is calculated (see Table II). Higher value of *GSim* means higher probability for the concrete service to substitute the abstract one.
     For similarity evaluation purposes, the ARTEMIS tool [8, 7] is used. In particular, *GSim* is calculated for a pair of service descriptors S_i and S_j as a weighted sum of an entity-based similarity coefficient $ESim(S_i, S_j)$ and a functionality-based similarity coefficient $FSim(S_i, S_j)$.

– *Entity-based similarity coefficient.* The Entity-based similarity coefficient of two descriptors S_i and S_j, denoted by *ESim*(S_i, S_j), is evaluated by comparing the input/output information entities contained in them.
In particular, names of input and output entities are compared to evaluate their degree of affinity $A()$ (with $A() \in [0, 1]$). The affinity A() between names is computed exploiting a thesaurus of weighted terminological relationships (e.g., synonymy, hyperonymy) supported by ARTEMIS.
Two names n and n' of entities have affinity if there exists at least one path of terminological relationships in the thesaurus between n and n' and the strength of path is greater or equal to a given threshold.
The higher the number of pairs of entities, one from the first service and one from the second, with affinity, the higher the value of $ESim$ for the considered services.
– *Functionality-based similarity coefficient.* The Functionality-based similarity coefficient of two descriptors S_i and S_j, denoted by *FSim*(S_i, S_j), is evaluated by comparing the operations contained in them. Also in this case, the comparison is based on the affinity $A()$ function.
Two operations have affinity if their names, their input information entities and output information entities have affinity in the thesaurus. The affinity value of two operations is evaluated on the basis of the affinity values of their corresponding elements in the descriptors. This evaluation is also used in the following step of operation matching. The value of $FSim$ coefficient is such that the higher the number of pairs of operations, one from the first service and one from the second, with affinity, the higher its value for the considered services.
As an example of operation similarity evaluation the reader should consider the operations:

```
generateMro
      input={businessName, fileInput}
      ouput={mroID, businessName}


createMro
      input={commercialName, inputFilePath}
      ouput={mroID, commercialName}
```

These two operations are similar according to our analysis, and have affinity 1 (see Table III), since: (i) value of A() is 1 for pair of names that are synonyms; (ii) the operation names are synonyms; (iii) for each I/O entity of `generateMro` there is a corresponding synonym entity in `createMro` and viceversa. A weaker correspondence between the names and the I/O entities of two operations would be evaluated with a lower value of affinity (as for `validateMro` and `checkMro` in Table III).

Table II. Descriptors matching table related to the MRO compatibility class

Concrete descriptor	GSim
Mro	1.0
MroUtilities	0.791

Table III. Operations matching table related to the MRO compatibility class

Abstract operation	Concrete operation	Affinity	Selected
generateMro	createMro	1.0	y
validateMro	checkMro	0.9	y
none	saveMro	0.0	n

Only the services with a value of *GSim* greater than a threshold t_c are selected for the next steps in which the mapping information will be generated. This threshold and more in general all the thresholds used in the Compatibility Evaluation Module are decided on the basis of experimentation and are proposed to the user of the VISPO platform that in any case can modify them to get better control on the compatibility evaluation process.

3. *Operation matching*: In this step, for each selected concrete service, for each operation required by the abstract service, the engine selects the operation of the concrete service with the highest affinity value. Specifically, the Compatibility Engine selects only the operations which have an affinity value greater than a given threshold t_{op}. As illustrated in the Table III, it can occur that the number of operations in the concrete and the abstract service are not equal. For instance, a concrete compatible service provides an operation not requested by the abstract service, or a concrete service provides less operations than the ones requested by the abstract service. Whereas in the former case the concrete service can be selected even if there is an operation that will never be invoked, in the latter the concrete service must be discarded because we assume that the minimal substitutable part is the service and not the operation.
4. *Input parameter matching*: In this step for each operation in the abstract service, the Compatibility Engine creates an affinity table in which all the input parameters are compared with the input parameters of a selected concrete operation. The relation between two parameters is considered to be valid if the affinity value is greater than the given threshold t_{ip} (see Table IV). If the number of the abstract

Table IV. Input parameters matching table related to the MRO compatibility class

Abstract parameter	Concrete parameter	Affinity	Selected
businessName	name	0.8	y
businessName	input	0	n
businessName	category	0	n
inputData	name	0	n
inputData	input	1	y
inputData	category	0	n
productCategory	name	0	n
productCategory	input	0	n
productCategory	category	0.8	y

parameters is greater or equal to the number of the selected concrete parameters the operations can be automatically substituted. Otherwise, if the number of concrete parameters is greater, the operation can still be used but it will be necessary, in the run-time phase, that the system asks to the cooperative process user to specify the input values for the extra parameters. In both cases the operation is accepted and the Compatibility Engine passes to the output parameter matching step.

5. *Output parameter matching*: The Compatibility Engine compares the output parameters of every abstract operation with the concrete ones of the selected concrete operation and for every comparison an affinity table is created. The relation between two parameters is considered to be valid if the affinity value is greater than the given threshold t_{op}. If the number of the abstract parameters is less or equal to the number of the selected concrete parameters the operations can be automatically substituted. Otherwise, if the number of concrete parameters is less, the concrete operation can not be used for service substitutability. In this case the concrete service must be discarded.
6. *Data matching*: This is the last step of the evaluation process. Given a pair of parameters identified in the previous step the Compatibility Engine analyzes their structure and evaluates their affinity. The affinity evaluation is performed only if both parameters have the same structure (i.e., both simple or both complex) while in other cases the concrete service is discarded. In the case of complex structures, the evaluation is performed by calculating the affinity between the simple types which compose them. The higher the number of simple types with affinity, the higher the complex structure affinity. As in previous steps, relations between parameters are considered valid only if their affinity value is greater or equal than a given threshold t_{sp}. The abstract service can be substituted by the analyzed concrete service only if all its parameters are related to the ones defined by the concrete specification.

3.2.3. Compatibility Module

The main purpose of the Compatibility Module is to implement the evaluation of the semantic service similarity coefficients as described in Section 3.2.2. As shown in Figure 9, the Compatibility Module is invoked by the Compatible Service Provider to which the Compatibility Module evaluations are returned. These evaluations are used at the publication phase by the domain experts to check the correctness of service assignments to compatibility classes. Furthermore, at the execution phase the semantic service similarities are taken into account to retrieve the most suitable services in a compatibility class for executing a required process activity.

In our architecture this module is realized by wrapping the ARTEMIS system [8, 7] that provides tools for analyzing the degree of semantic similarity among descriptors, and for the matching of classes of information, on the basis of ontological information provided by a thesaurus. The ARTEMIS system is provided with a CORBA interface and so, as shown in Figure 9, the Compatibility Module is invoked by the Compatible Service Provider using CORBA communication.

ARTEMIS is therefore used in the context of the VISPO system to evaluate descriptor similarity and, more in general, terminological affinities. The first functionality allows the Compatible Service Provider to compare service descriptors in order to evaluate semantic similarity coefficients for the analyzed services. The term affinity evaluation functionality is instead used for comparing the terms occurring into the interfaces of the services for the purpose of constructing the mapping information between abstract and concrete services, that the Compatible Service Provider proposes to the domain expert as wrapper skeletons on which complete wrappers are built on (See Section 3.3).

For the similarity and affinity computation, ARTEMIS exploits the terminological knowledge provided by a domain dependent thesaurus, the role of which is to provide knowledge about domain specific terms according to semantic relationships such as *synonym-of*, *broader-than*, and *related-to*. In particular, in our experimentation the thesaurus is filled with domain knowledge provided by domain experts.

This module is not directly used by the Compatibility Engine, but it is accessed by the Compatibility Evaluation Module which uses the affinity term evaluation functionality to built more complex functions.

3.2.4. Invocator

The main purpose of the Invocator, once the mapping information, the WSDL description of the abstract service, and a list of compatible services are given, is to manage the concrete services invocation. As described in Section 3.1.4, the Invocator receives an abstract invocation request based upon an abstract service definition, selects a concrete service, invokes the service through its wrapper and returns the response as defined by the abstract interface definition.

The Invocator architecture, depicted in Figure 10, comprises four main modules:

- *Invocator HTTP Interface*: the methods exposed through this interface allow *linking* an abstract service, *invoking* an abstract service, *unlinking* from an abstract service, and *retrying* an abstract service previously linked (see Section 4).
- *Invocator Engine*: it is the core of the Invocator and contains the application logic of the module. Given an abstract service definition and a list of compatible concrete services ordered by affinity degree, it starts to invoke the service with the greater affinity value. If this invocation fails, it tries the second service in the list and so on. The invocation process is started retrieving the wrapper designed for the selected concrete service and passing it to the Service Invocator module.
- *Service Invocator*: this module invokes concrete services. It is activated by the Invocator Engine and performs the invocation using the assigned wrapper.
- *Wrapper Repository*: this repository contains all the wrappers associated to the concrete services. Wrapper are created in a semi-automatic way. Using the mapping information generated by the system, domain experts can build wrappers and save them into this repository. When the *invoke* operation is performed, the Invocator Engine retrieves the correct wrapper from the repository and plug it into the Service Invocator.

3.3. Wrappers Creation

As stated in 3.1.4, wrappers are created by domain experts starting from mapping information. Given an abstract service, a concrete compatible service and related mapping information, domain experts have to implement the specific interface that defines wrapper methods.

Domain experts implement the logic of methods, which allows translating abstract service invocations into concrete service invocations. These methods, used by the Service Invocator to perform effective service invocation, allow the system to:

- locate concrete service access points;
- translate abstract operations inputs into concrete operations inputs;
- translate concrete operations outputs into abstract operations outputs.

In some cases may it may occurs that a concrete operation requires more input parameter than the abstract one. This means that it is not possible to invoke this operation only with an abstract parameters translation, but it is necessary to retrieve all missing parameters by asking them to the cooperative process executor. The domain expert is in charge to create wrappers able to retrieve additional needed information.

In our prototype implementation, we request lacking parameters to process executors via e-mail, redirecting them to an HTML page where the requested values can be inserted.

4. Cooperative Processes and Flexible Invocation

The execution of a cooperative process requires sophisticated interactions among services. In the VISPO system this task is demanded to the Orchestration Engine. This module provides

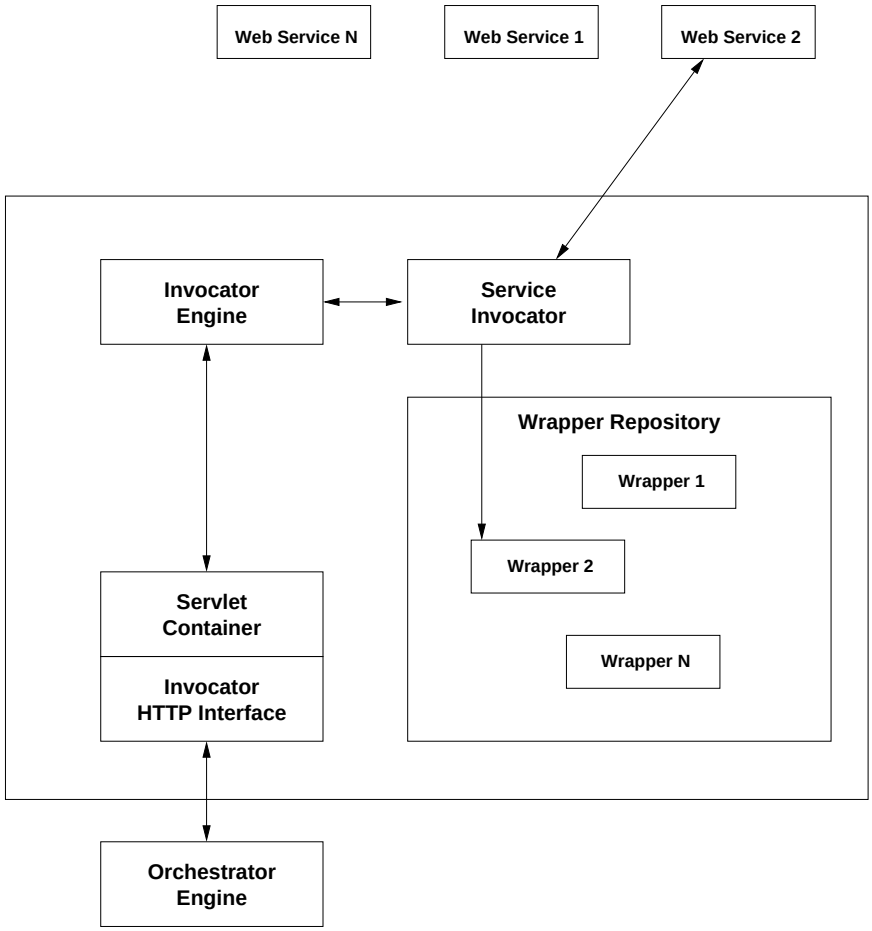

Figure 10. Invocator Architecture

workflow functionalities, that drive the Invocator in order to correctly manage the information exchanges between services.

The Orchestration Engine operates on the basis of the abstract interface definition of the involved services, and the instantiation between abstract services and real services is automatically managed by the Invocator. In particular, when the Orchestration Engine is initiating the execution of a new process instance of the workflow, it asks the Invocator to link (i.e., to associate to the current process instance) specific concrete service instances of the required abstract services, and during the workflow invokes their operations; if, in some points, a service is unavailable, the Invocator automatically substitutes it with a compatible

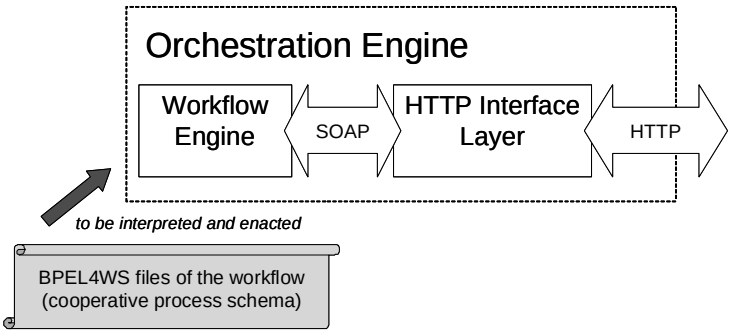

Figure 11. Orchestrator Architecture

one, and the Orchestration Engine is shielded by such a substitution. When the Orchestration Engine has finished with a service, it explicitly unlinks it, in order to let the Invocator release its resources (unplug wrappers, etc.).

The communication between the Orchestration Engine and the Invocator is based on HTTP protocol, in order to avoid incompatibilities due to different SOAP implementations that we experimented during the development of the system.

From a methodological point of view, a cooperative process is designed by mean of a Petri Nets-based formalism, as described in [19]. The use of such a precise formalism allows the verification of correctness properties, such as the absence of deadlocks and the consistency of each service at the end of the cooperative process enactment. The Petri Nets-based schema of the cooperative process is then translated to an effective orchestration technology in order to enact process instances. Among the many languages that have been proposed for orchestration [28], we concentrated on BPEL4WS [15]; the Petri-Net is therefore translated into two BPEL4WS compliant files, one for the orchestration interface and one for the orchestration implementation [17]. Such files are the process schemas to be enacted by the Orchestration Engine.

Figure 11 shows the internal architecture of the Orchestration Engine. It is composed by two main components, the Workflow Engine and the HTTP Interface Layer.

Workflow Engine. This module interacts with the HTTP Interface Layer by mean of SOAP messages[§]. In order to develop the workflow engine, we experimented the currently most widespread BPEL4WS compliant engines, BPWS4J [2] and Collaxa BPEL Server [10], finally adopting the Collaxa BPEL Server 2.0.

[§]In BPEL4WS, an orchestration is itself a Web service, and therefore communicates through SOAP messages. As we decided that the architecture is completely HTTP based, we need to "wrap" such a protocol.

```
<process name="VispoDemo"
         suppressJoinFailure="yes"
         targetNamespace="http://progettovispo.com"
         xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
         xmlns:tns="http://progettovispo.com"
         ........

  <partnerLinks>
    ......
    <partnerLink name="HTTPInterfaceLayer"
                 partnerLinkType="cdp:HTTPInterfaceLayer"
                 partnerRole="HTTPInterfaceLayerService"/>
    ......
  </partnerLinks>

  <flow>

     <sequence>
        <assign>
            .......
            <copy>
               <from expression="'Invoke'"/>
               <to part="operazioneInvocator" variable="invocationGMC"/>
            </copy>
            <copy>
               <from part="ServiceName" variable="InvocatorResponseS1"/>
               <to part="ServiceName" variable="invocationGMC"/>
            </copy>
            .......
        </assign>
        <invoke name="invocaGMC"
                partnerLink="HTTPInterfaceLayer"
                portType="cdp:HTTPInterfaceLayer"
                operation="invocation"
                inputVariable="invocationGMC"
                outputVariable="responseGMC"/>

     .........

     </sequence>

  .........

  </flow>

  .........
</process>
```

Figure 12. BPEL file example

In Figure 12 we show a fragment of the final BPEL4WS orchestration implementation of the cooperative process described in Section 2. As we can see, the Workflow Engine always invokes the HTTP Interface Layer (which is itself a Web service) and this layer is in charge to send to the Invocator (through HTTP calls) all the information needed to correctly invoke services, by using only the abstract interface of the service. Specifically, the application interface between the Orchestration Engine and the Invocator consists of the following operations:

- `link(abstractService) return serviceIdentifier`: the Orchestration Engine requires the Invocator to establish a connection with the specified

`abstractService`. The Invocator returns the `serviceIdentifier` to use for service invocation.
- `invoke(serviceIdentifier, operation[p1, ... pn]) return operationResult`: when the Orchestration Engine needs to invoke an `operation` with parameter `p1`, ... `pn`, obtaining the `operationResult`.
- `unlink(serviceIdentifier)`: when the Orchestration Engine releases a service that will be no more used in the process.
- `retry(serviceIdentifier)`: when the Orchestration Engine wants to test the availability of a service.

HTTP Interface Layer. This is the component responsible to interface the Workflow Engine with the Invocator module. It elaborates the SOAP messages received in order to send the correct instructions to the Invocator. The HTTP Interface Layer receives the Orchestration Engine directives and convert them into HTTP messages; then it converts responses received in HTTP back to SOAP. It can be viewed as a wrapper service between the HTTP protocol used by the Invocator and the SOAP protocol used by the Workflow Engine.

5. Performance Evaluation

A prototype of the architecture has been developed in the context of the VISPO project and a demo of its functionalities is available on line at the URL: `http://cube-si.elet.polimi.it:8082/vispo/index.html` in which a cooperative process is offered and it is possible to test the substitution feature.

Although our implementation is still a prototype and more work is needed to make the VISPO system stable and efficient, we have made some performance tests. In these tests we have measured the response time for the Invocator and the Compatible Service Provider.

Evaluations were made separately because the compatibility evaluation function provided by the Compatible Service Provider is used only during the publication phase of concrete services while the Invocator functionalities are used during the process execution phase. In this sense, we could say that the Compatible Service Provider does not have any direct impact on the performance of the VISPO system during the execution of cooperative processes.

5.1. Test Environment

In order to study the performances of both Compatible Service Provider and Invocator modules, we designed a test environment to simulate the usage of our system.

The test environment was composed by 7 dedicated box connected with a 100 Mbit LAN. The VISPO system was installed on a bi-processor Intel Xeon 2.4 GHz, 1 GB RAM, Win2000 server and deployed using Tomcat 4.1.27 (200 threads), while each one of the Web services used for composing the test cooperative processes was deployed on a different workstation (processor Intel PIII, 256 MB RAM, SuSE 9.0). To avoid the effects of other platform components, Web service methods perform no business logic but simply return a fixed result.

For coding the test clients we have used TestMaker [24]. TestMaker provides an environment for building software to test Web services and Web applications. We have built two types of test application for testing the Invocator and the Compatible Service Provider separately.

In the following we describe how tests have been performed and report some testing results.

5.2. Invocator Performances

The main goal of the Invocator module is to manage the invocation of the abstract services that compose a cooperative process. Given an abstract service, an abstract invocation request, an ordered list of compatible concrete services, and a set of wrappers, the Invocator is responsible for the management of the dynamic abstract services invocation. We have performed two kind of tests, the first for relating the availability of the process with the Invocator response time, and the second for evaluating the average response time of the Invocator under heavy load conditions.

5.2.1. Process availability and invocator response time

For relating the process availability with the Invocator response time we have used a simple test process. The process comprises a sequence of five equivalent activities performed using Web services with the same interface, delivered by different service providers, and executed on different servers (i.e., Web services execution are performed in an independent manner). The process ends successfully only if all its activities are performed without failures. Anyway, the simplicity of the process does not influence the performance evaluation of the Invocator module, which is only responsible for the invocation of abstract services and not for the orchestration of cooperative processes.

Obviously, our methodology for managing the execution of cooperative processes is strictly depending on the availability of the involved services. If the Web service executions are independent one from the other, and if they have the same failure rate, the availability (Av) of the test process can be calculated using the formula:

$Av = (1 - P(WSF))^5$, where $P(WSF)$ is the probability of a concrete service failure.

Let us suppose that all Web services used during the execution of the process are retrieved from the WWW and that their failure rate is 16% [18]. In the case there is only one concrete service for every abstract service, the service substitution between concrete services cannot be performed and the value of the process availability is $Av = 0.4182\%$. Increasing the number of the concrete services that can substitute an abstract service will reduce the probability $P(SWF)$, increasing the availability Av of the process. Figure 13 reports the relation between the process availability and the number of concrete services available for each abstract service. Even this evaluation is done using independent Web services with a failure rate value of 16%.

The Invocator module performs the invocation of concrete services using wrappers for transforming abstract parameters into concrete parameters and vice-versa. This means that the time spent by a wrapper for translating the parameters could be relevant with respect to the Invocator response time. Wrapper response time depends on the complexity of the wrapper

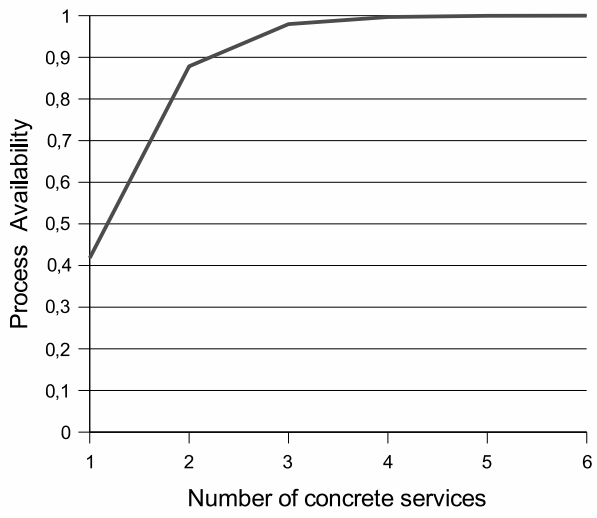

Figure 13. Process availability

itself, more complex are the transformations performed by the wrapper and higher will be the response time.

In order to evaluate the Invocator response time, we have conducted two tests using a client deployed on single workstation (Processor Intel PIII, 256 MB RAM, SuSE 9.0). Each test simulates the invocation of an abstract service, with invocation of one to six different alternative services, where we have supposed that the difference between the abstract and concrete services resides only in the name of the invoked operation and in the name and in the types of the exchanged parameters. In the first test we have used one operation with one parameter, while in the second we have used one operation with ten parameters.

The graphics of Figure 13 and 14 put in evidence that the response time increases linearly with the number of concrete services while the availability of the process increases exponentially.

Obviously, the response time of the Invocator does not depends only on the response time of the wrappers. There are two other factors that have an impact on the Invocator performance: the time spent for retrieving a wrapper from the repository and the time spent by the Invocator for managing the invocation request.

Given an abstract service invocation, the mean time spent by the Invocator for managing the request is $20ms$ while the mean time spent for retrieving a wrapper is $47ms$. It should be noticed that for every invoked concrete service, a wrapper must be retrieved.

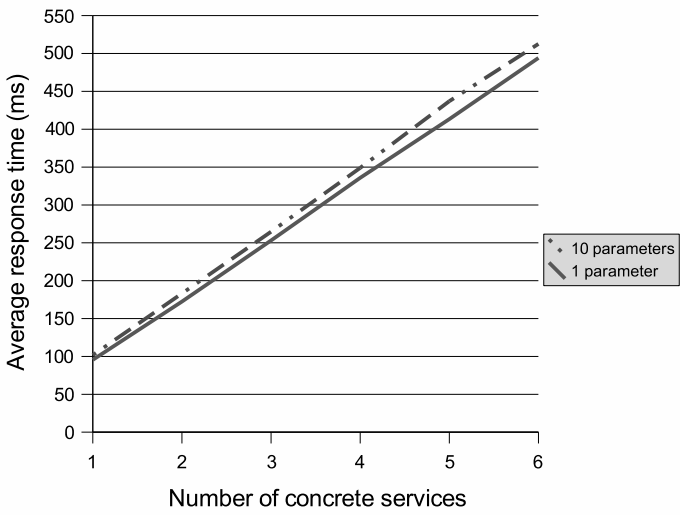

Figure 14. Invocator response time using 1 or 10 parameters

5.2.2. Invocator performance under heavy load conditions

The Invocator performances have been tested using five clients that execute the same cooperative process (see Figure 15) concurrently. Each client has been deployed on a different workstation (Processor Intel PIII, 256 MB RAM, SuSE 9.0). During this test we have used a different process in order to evaluate the average response time of the Invocator without the concrete service substitution but using different kinds of wrappers. The process is quite simple and it contains sequential activities performed by different kinds of services.

In order to evaluate the average response time of the Invocator under growing load conditions, we initially performed a cycle of 10 cooperative process executions on the first client. Successively, the test has been repeated running two clients concurrently where each client performed a cycle of 10 cooperative processes executions. Test has been repeated up to 5 clients running concurrently. Figure 16 reports experimental results.

The entire test was repeated using cycles composed of 100 process executions in order to evaluate the scalability of the Invocator module and results are reported in Figure 17.

The graphic depicted in Figure 18 relates the results of the performed tests comparing the average response time of one process execution measured for 10 and 100 cycles.

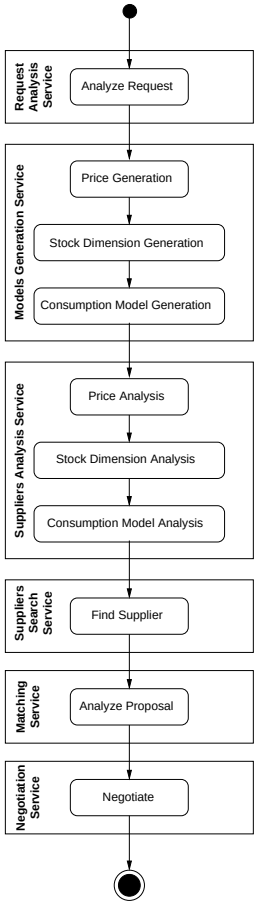

Figure 15. Test process

5.3. Compatible Service Provider Performances

The goal of the Compatible Service Provider is to compare Web services, evaluate their affinity and create mapping information for building service wrappers.

Given two or more Web services, mapping information is created. The evaluation of their affinity is performed, and it has to be recalculated only if at least one of the interfaces of the involved Web services has changed.

The main factor that influences the performance of the Compatible Service Provider is the WSDL structure of the Web services on which the compatibility evaluation is performed. For

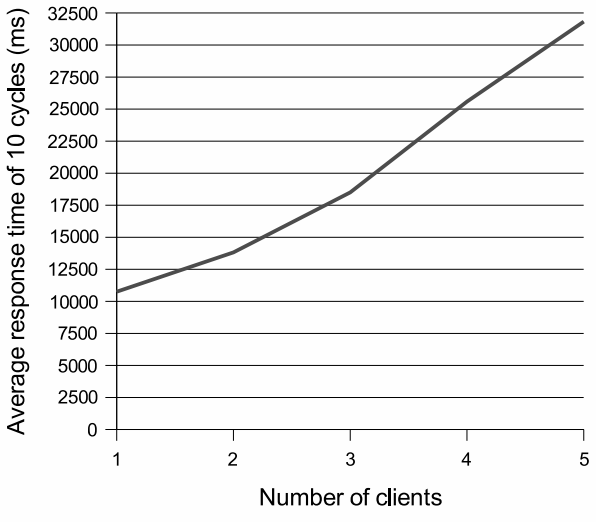

Figure 16. Invocator average response time measured on 10 cycles

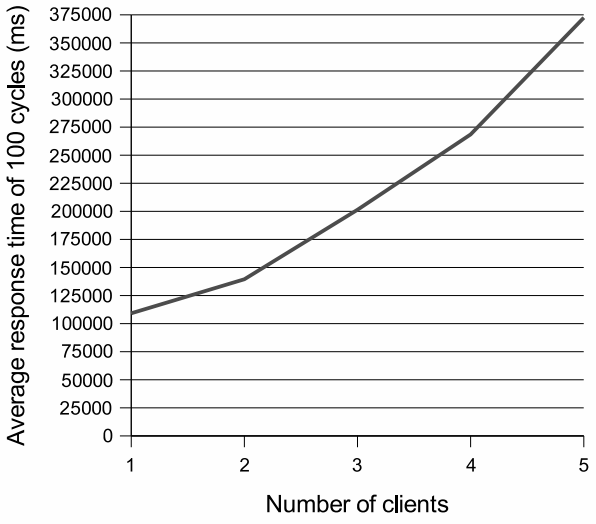

Figure 17. Invocator average response time measured on 100 cycles

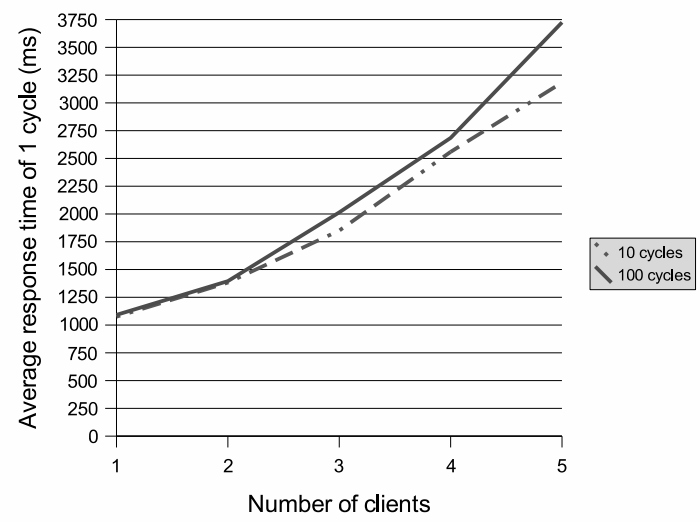

Figure 18. Invocator average response time of 1 process execution measured on 10 and 100 cycles

this reason, we conducted a test for evaluating the response time of the Compatible Service Provider considering different kinds of WSDL.

The test was conducted considering couple of Web services and using a client built with TestMaker and deployed on a single workstation (Processor Intel PIII, 256 MB RAM, SuSE 9.0). Their WSDL descriptions were made so as to increase the differences between the descriptors, augmenting the operations performed by the Compatible Service Provider for evaluating their compatibility (see Subsection 3.2.2). Each couple of WSDL documents contains the same number of operations and parameters. The names of the operations and parameters are different and all parameters are simple types.

In order to evaluate the impact of the number of operations and parameters on the Compatible Service Provider performances, we have made three kinds of tests. The first test evaluates the Compatible Service Provider average response time using couple of WSDL with one to five operations, where each operation has one parameter. The second test uses couple of WSDL with one to five operations and five parameters for each operation, while the third test still uses couple of WSDL with one to five operations, but with ten parameters for each operation. In Figure 19 experimental results are reported.

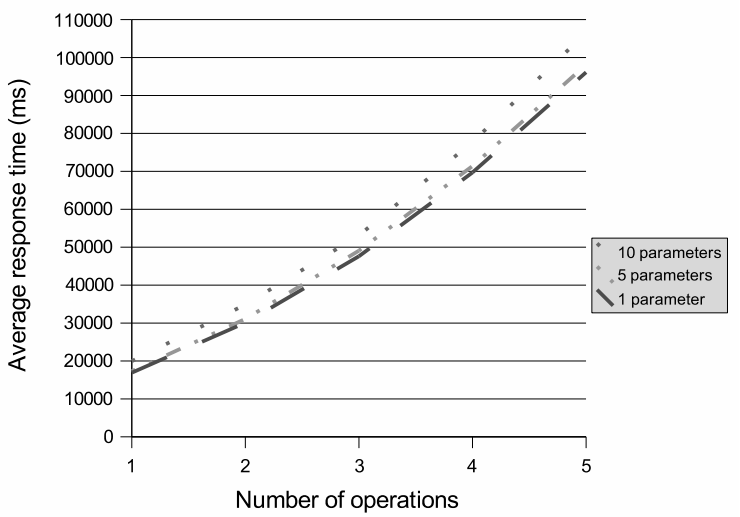

Figure 19. CSP average response time measured increasing the operations number

6. Concluding Remarks

Dynamic invocation of Web services is a very important task in cooperative processes execution.

In this paper, we have introduced an architecture for Web service compatibility, based on interface analysis. The architecture allows managing cooperative processes in a very dynamic way, allowing Web services substitution during the run-time phase.

In the future, we will explore how to consider other aspects of Web services during the compatibility evaluation phase, including quality of service aspects and semantic interface evaluation. Other improvements will regard the ability of our methodology to perform more sophisticated affinity evaluation in order to perform affinity evaluation on composed elements too (e.g., one operation affine with the composition of two ore more other operations), overcoming the 1:1 service mapping behavior.

In addition, the prototype will be improved adding the ability to perform asynchronuos calls to Web services.

ACKNOWLEDGEMENTS

Part of this work has been supported by MIUR, through the "Fondo Strategico 2001" project *VISPO* and the "FIRB 2001" project *MAIS*.

REFERENCES

1. A.Brown and H. Haas. Web services glossary. Technical report, W3C, http://www.w3.org/TR/ws-gloss/, February 2004.
2. IBM alphaWorks. BPEL4WSJ - Business Process Execution Language for Web Services Java Run Time. `http://www.alphaworks.ibm.com/tech/bpws4j`, 2003.
3. A. Arkin, S. Askary, and S. Fordin et al. Web Service Choreography Interface (WSCI) 1.0. W3C Note. `http://www.w3.org/TR/wsci/`, 2002.
4. Assaf Arkin. Business process modeling language, 2002.
5. Toufic Boubez and Luc Clment. Uddi tmodels: Classification schemes, taxonomies, identifier systems, and relationships, version 2.04, December 2002.
6. P. Brittenham, F. Cubera, D. Ehnebuske, and S. Graham. Understanding WSDL in a UDDI Registry. `http://www-106.ibm.com/developerworks/webservices/library/ws-wsdl/`, September 2002.
7. S. Castano and V. De Antonellis. A schema analysis and reconciliation tool environment for heterogeneous databases. In *Proceeding of IDEA'99 Int. Database Engineering and Application Symposium*, Montreal, Canada, August 1999.
8. S. Castano, V. De Antonellis, and S. De Capitani di Vimercati. Global Viewing of Heterogeneous Data Sources. *IEEE Transactions on Knowledge and Data Engineering*, 13(2), 2001.
9. J.Y. Chung, K.J. Lin, and R.G. Mathieu. Web services (special issue). *IEEE Computer*, 36(11), October 2003.
10. Collaxa. BPEL Server, 2003.
11. E. Colombo, C. Francalanci, B. Pernici, P. Plebani, M. Mecella, V. De Antonellis, and M. Melchiori. Cooperative Information Systems in Virtual Districts: the VISPO Approach. *IEEE Data Engineering Bulletin*, 25(4), 2002.
12. OASIS/ebXML Registry Technical Committee. *OASIS/ebXML Registry Information Model v2.5*. OASIS, http://www.oasis-open.org/committees/regrep/documents/2.5/specs/ebrim-2.5.pdf, June 2003.
13. E. Damiani and M. G. Fugini. Fuzzy Identification of Distributed Components. In *B. Reusch ed. Proceedings of the 5th Fuzzy Days International Conference, LNCS 1226*, 1997.
14. V. De Antonellis, M. Melchiori, B. Pernici, and P. Plebani. A Methodology for *e*-Service Substitutability in a Virtual District Environment. In *Proceedings of the 2003 Conference on Information Systems Engineering (CAiSE 2003)*, Velden, Austria, 2003.
15. S. Thatte (ed.). Business Process Execution Language for Web Services Version 1.1. Document. `ftp://www6.software.ibm.com/software/developer/library/ws-bpel11.pdf`, 2003.
16. Roberto Chinnici et al. *Web Services Description Language (WSDL) 1.1.* W3C, http://www.w3.org/TR/wsdl, March 2001. W3C Note.
17. E. Felici. *Orchestrazione di Processi Cooperativi.* Tesi di Laurea in Ingegneria Informatica, Università di Roma "La Sapienza", Facoltà di Ingegneria, 2003 (in Italian) (the thesis is available by writing an e-mail to: `mecella@dis.uniroma1.it`).
18. Su Myeon Kim and Marcel Catalin Rosu. A survey of public web services. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 312–313. ACM Press, 2004.
19. M. Mecella, F. Parisi Presicce, and B. Pernici. Modeling *e*-Service Orchestration Through Petri Nets. In *Proceedings of the 3rd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2002)*, Hong Kong, Hong Kong SAR, China, 2002.
20. M. Mecella and B. Pernici. Building Flexible and Cooperative Applications Based on *e*-Services. Technical Report 21-2002, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Roma, Italy, 2002 (available on line at: `http://www.dis.uniroma1.it/~mecella/publications/mp_techreport_212002.pdf`).
21. M.P. Papazoglou and D. Georgakopoulos. Service Oriented Computing (Special Issue). *Communications of the ACM*, 46(10), 2003.
22. C. Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(11), October 2003.
23. T. Pilioura and A. Tsalgatidou. *e*-Services: Current Technologies and Open Issues. In *Proceedings of the 2nd VLDB International Workshop on Technologies for* e-*Services (VLDB-TES 2001)*, Rome, Italy, 2001.
24. PushToTest. TestMaker 4.1, 2004.
25. UDDI Committee Specification, http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.pdf. *UDDI Version 2.03 Data Structure Reference*, July 2002.
26. UDDI Committee Specification, http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf. *UDDI Version 2.04 API Specification*, July 2002.

27. UDDI Committee Specification, http://uddi.org/pubs/uddi-v3.0.1-20031014.pdf. *UDDI Version 3.0.1 API Specification*, October 2003.
28. W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1), 2003.