Provisioning of Complex Adaptive Services

L. Baresi, F. Daniel, A. Maurino,
S. Modafferi, E. Mussi, B. Pernici
Politecnico di Milano
P.zza L. da Vinci 32
20133 Milano, Italy
mussi@elet.polimi.it

D. Bianchini, V. De Antonellis
Università degli Studi di Brescia
Via Branze 38
25123 Brescia, Italy

ABSTRACT

Service oriented computing is becoming the standard paradigm to support the creation of applications composed of e-services selected from a registry. Nowadays, we are assisting to the proliferation of standardized approaches to describe such services, but there is the general agreement of distinguishing the general characteristics of services from the characteristics linked to service invocation. In many case, the selection of services is static and based on matching techniques to retrieve the most appropriate service.

The paper presents the MAIS architecture to provide highly adaptive e-services in a mobile and interactive environment. It focuses on service selection and invocation, context-aware orchestration, and mechanisms for managing user interaction in a service oriented architecture. We propose adaptivity at different levels: at the process level, but also at the level of the selection of a concrete service given an abstract description. Selection is based on suitable ontologies and can consider the actual context and user characteristics to retrieve the most suitable services. The paper describes the main components of the architecture and exemplifies them on a simply process for a shipping company.

Keywords

Novel architectural approaches for service-oriented computing, Service and Mobile Computing Service description and advertisement, Service discovery and selection, Location-based services, Mobile e-businesses, Core service activities and technologies

1. INTRODUCTION

The emerging paradigm of service oriented computing [**?**] supports the creation of applications by composing e-services selected from a registry among a variety of available services with given characteristics. E-services may be invoked directly by the application in which they are used. Essential to this paradigm is the definition of e-services using a standardized approach; in the literature, proposals of service description languages, such as WSDL, of service ontologies, such as in AgFlow [22], or of semantic web services [1] are all going towards the direction of separating general characteristics of services from the characteristics linked to service invocation with a given protocol. In the above mentioned approaches, service selection is generally static, assuming matching techniques to retrieve the most appropriate service. In VISPO [2], the authors introduced the concept of concrete and abstract services in the context of process definition, allowing the designer to specify the process in terms of abstract services and then providing an invocation environment to select the most appropriate service. In the selection and execution, they evaluated the availability of the selected process and provided mechanisms for substituting services whenever they are not available.

However, both in VISPO and AgFlow, where service unavailability is considered at run time, the assumption is that, beyond unavailability of services, the context of invocation is always the same. This assumption cannot be considered valid anymore in applications running in a very variable environment in terms both of architecture and its components. In such environments, for example, in the case of mobile information systems [13], services invocation may vary depending on their availability over the network, on parameters of devices on which they are invoked that influence their quality, and on the characteristics of the networking infrastructure. In addition, e-services might be used in the process several times and their execution environment might vary over time.

The goal of this paper is to propose the MAIS architecture, along with its mechanisms, to design and execute complex services composed of adaptive services. We propose adaptivity at different levels: at the process level, at the level of selection of a concrete service for a given abstract service, in the user environment. We support service selection with an enhanced UDDI registry, storing descriptions of abstract and concrete services, including information about quality parameters on the provider side. The proposal has been developed in the MAIS (Multichannel Adaptive Information Systems) Project [20].

The rest of this paper is organized as follows. Section 2 presents a running example of mobile information systems for a shipping company to provide motivations for the aspects discussed in the rest of the paper. Section 3 describes the MAIS functional architecture, focusing on orchestration and concrete service selection and invocation. It also introduces the basic ontology of services of the MAIS Registry.

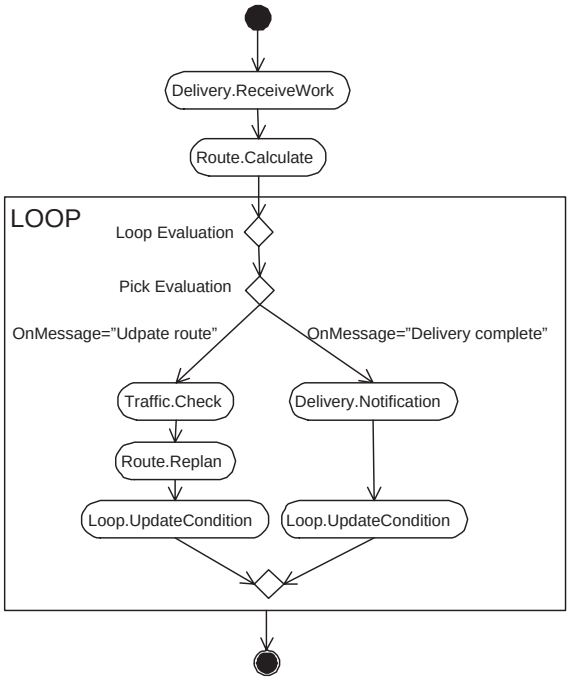

Figure 1: Workflow of shipping assignment and execution phases

Finally, it discusses a mechanism for decoupling e-service invocation from the design of the user environment in terms of its interaction with the system, based on an extended WebML model [9]. Section 4 discusses our proposal in relation to the state of the art.

2. THE SHIPPING COMPANY

This section presents the running example used throughout the paper. It describes the typical problems of a shipping company that wants to optimize the delivery of packages. We concentrate on a simplified version of the process to deliver goods and imagine that the need for optimization and adaptation of the delivery procedure to the context/profile leads to enacting the process in several different ways.

ShipEveryWhere, our shipping company, has a unique process to support the delivery of all packages. for simplicity, we consider a single item and we also assume that he process starts after assigning the item to the best vehicle. The process includes the creation and update of the route followed by the vehicle. Figure 1 shows the process that oversees all the phases. Notice that we use a dot notation to name activities: The first part identifies the service, while the second part specify the operation.

The process starts with the assignment of the task to the vehicle that carries the item (service *Delivery*, operation *Receive Work*). The *ShipEveryWhere* control center, through an appropriate user interface, assigns the item and relative delivery information to the selected driver. According to the destination and dimension of the item, the vehicle can be a truck for destination longer than 200 kilometers, a van for destination between 10 and 200 kilometers and bulky packages, and a motorbike for close destinations, i.e, less than 10 kilometers and small boxes. Each vehicle is equipped with a device to interact with the control center via a GPRS interface. In particular, each truck hosts a laptop; vans have PDAs, and who drives a motorbikes uses a smart-phone. Vehicles are equipped with different devices because of the different uses and the room on board. This choice, however, implies that the enactment of the process varies and must cope with the device on the actual vehicle.

After the assignment phase the driver calculates the best route to deliver the item (service *Route*, operation *Calculate*). This activity varies according to capabilities of the device hosted by the vehicle and can be done in different ways. For example, the control center can send required data, the vehicle can use local (context-dependent) services to discover traffic conditions and calculate the best option, or drivers can decide based on their own experience.

While driving towards destination, the driver can either notify the control center that the item is delivered (service *Delivery*, operation *Notification*) or request the current situation of traffic condition and consequently replan the route. The traffic update can be required by the driver, in case of heavy traffic on the selected route or by the control center to notify the driver of congested traffic conditions on the route (message *Update Route* of *Pick* activity). The selection of the actual services that detect traffic conditions and replan the route depends on the driver's profile (e.g. the used device) and the specific context in which the request is place (e.g. the availability or absence of a GPRS network can affect the set of available alternatives). If the driver is not able to connect to the control center, due the absence of GPRS signal or because the bandwidth is too low, he cannot connect to the *TIER* service (**T**raffic **I**nformation on **E**uropean **R**oads). If the vehicle cannot use GPRS channel (or if the driver declares in his profile that he does not want to pay for it) the replanning must be done locally (that is, manually) with the information on board or by using the driver's experience. In this last case, data must be supplied by the driver by means of a special-purpose user interface.

3. FUNCTIONAL ARCHITECTURE

This section introduces the MAIS architecture, its components, and the relationships among them.

3.1 MAIS services

Before introducing the actual architecture, we must set the jargon used in the paper and clarify that we distinguish between:

- *concrete services*, which are directly invocable services, with public WSDL interfaces and bindings to specific implementations;
- *abstract services*, which are services that cannot be invoked directly. These abstract descriptions are extracted – by means of integration mechanisms – from a set of existing concrete services clustered on the basis of their functional similarity (see [4]). Their capabilities, represented using a WSDL interface, are representative of the capabilities of services in the cluster. In particular, the set of defined capabilities is "minimal" in that, for example, the set of operations in the abstract service interface are only those common to all the services in the cluster. The designer can possibly force further capabilities considered relevant to the cluster, for example, because they are present in most services of the cluster. In any case, proper mapping

rules between the capabilities in the abstract service and those in the corresponding concrete services must be defined. They are defined on operation names and on I/O entity names and are tables (one for each abstract capability).

Services are characterized by means of semantic information about functional, context, and quality aspects:

- The functional description is given in terms of the operations that the service performs, the input entities it requires for the execution, and the output entities it produces after execution. Constraints on input and output entities can be specified;
- The context description is given in terms of conditions under which the service is provided and used. In particular, it refers to the `Channel` used for service provisioning (modeled by means of `Device`, `Network`, `NetworkInterface` and `ApplicationProtocol`), `Location` and `Time` information;
- the quality of service is expressed by means of a set of standard quality dimensions, imposed by the channels used for service delivery and guaranteed by the service provider; each dimension is described by a name and a range of admissible values.

The language proposed for describing services is WSDL properly augmented to represent specific semantic information. In particular, context information and constraints about input and output entities are expressed by means of pre- and post-conditions, i.e., logical expressions (represented as conjunction of pairs <`element,value`>) that must be satisfied before (pre-conditions) and verified after the service execution (post-conditions).

3.2 MAIS architecture

Figure 2 shows the MAIS functional architecture and the relationships among its modules. The architecture is composed of six modules that cooperate to manage and provide complex e-services in a context aware manner.

The architecture considers two different classes of users: *Designers*, who create and publish the MAIS services, and *End Users*, who use the architecture to find and use the published services.

While *Designers* access directly the platform through the *MAIS Registry*, the access for *End Users* is mediated by two modules, the *User Environment* and the *Platform Invocator*. The goal of these two modules is to provide the functionality that allow *End Users* to interact with the MAIS architecture. These features are not only limited to the search and invocation of services, but also comprise the management of tasks related to the execution of complex services.

The *Platform Invocator* represents the point of contact between *End Users* and the MAIS architecture and hides the complexity of the architecture. Its interface exports a series of operations, which allow to interact programmatically with the architecture, performing operations like: i) search published services in the *MAIS Registry* ii) execute the chosen services, and iii) manage the interaction with *End Users* during the execution of a complex service.

Obviously, such components prevent *End Users* from using the MAIS architecture in an interactive way. For this reason, *End Users* interact through the *User Environment* and not directly with the *Platform Invocator*. The *User Environment* provides the graphical interface for the *End Users* who want to interact with the MAIS architecture. What distinguishes this module from a simple static GUI is the ability to dynamically generate the user interface with respect to the context in which *End Users* are (i.e., device, available communication protocols, user profile). The *User Environment* provides the same functionality as those exported by the *Platform Invocator* but through a graphical and context-aware interface. The information about the context is taken from the *MAIS Reflective Architecture Interface*.

This module represents the access point to the reflective middleware and allows the other modules of the architecture to observe and modify the execution context and capture relevant events from the reflective middleware. These events provide useful information to the architecture for adaptive service provisioning. (i.e., QoS degradation or battery level of a mobile device).

Once *End Users* have selected and invoked a service using the *User Environment* and the *Platform Invocator*, the management of such an execution is performed by the core modules of the MAIS architecture. These modules are: i) the *Process Orchestrator* for managing the execution of complex concrete services and ii) the *Concrete Service Invocator* for instantiating the services and executing the calls of concrete services operations.

The *Process Orchestrator* is used when a complex service is executed and its process has to be orchestrated. It manages the process state and, step by step, interacts with the *Concrete Service Invocator* for invoking each operation specified in the workflow definition. The *Process Orchestrator* invokes abstract operations using abstract parameters; it is up to the *Concrete Service Invocator* to translate abstract parameters into concrete ones and invoke the concrete operation compatible with the abstract one. Another task performed by the *Process Orchestrator* is the delivery of process activities to *End Users*. Process activities are defined by the *Designer* during the definition of the workflow and delivered to users at run time. In order to perform this task, the *Process Orchestrator* uses specific capabilities provided by the *Concrete Service Invocator*.

As stated before, besides the orchestrator, there is another module that is needed to invoke concrete services: the *Concrete Service Invocator*. This module is in charge of managing the invocation of services and delivering activities (tasks) to *End Users*.

The *Concrete Service Invocator* delivers activities to the assigned *End User* and, in the case of a service invocation, it can:

- Start the invocation of services. When the *Platform Invocator* asks for a service invocation, the *Concrete Service Invocator* invokes the right concrete service after the interaction with the registry, if needed.
- Invoke abstract operations. This is a sophisticated functionality used by the *Process Orchestrator* for invoking abstract operations. An abstract service cannot be invoked and so, an initial phase for selecting concrete services is needed. During this phase, called *link phase*, the *Concrete Service Invocator* accesses the *MAIS Registry* for finding concrete services and evaluate their affinity with respect to the abstract service (i.e., the request). Once a compatible concrete

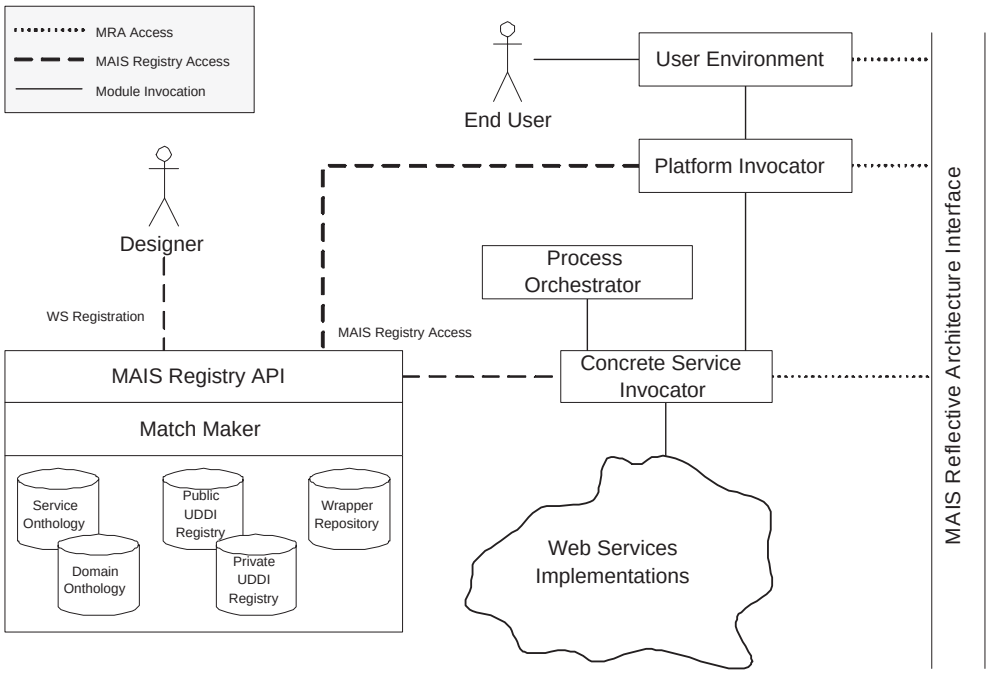

Figure 2: MAIS Functional architecture

service is chosen, the *Concrete Service Invocator* proceeds by invoking the concrete operation. The *Concrete Service Invocator* receives as input the parameters of the abstract operation, translates them into the concrete ones, invokes the concrete operation and then translates the concrete output parameters into abstract ones. The translation of parameters is performed by using wrappers registered in the *MAIS Registry*.

- Invoke concrete service operations. This invocation is made by accessing directly the concrete implementations of services and invoking the concrete operation by passing concrete parameters.

The last module of our architecture is the *MAIS Registry*. This module represents the registry of the MAIS architecture and contains suitable descriptions of all published services, along with other support information.
Services are published in the MAIS registry that is composed of: a *UDDI registry*, where services are registered with associated keywords, a *domain ontology*, where semantic information for service input/output annotation is maintained, and a *service ontology*, where services and semantic relationships among them are organized in two different layers (concrete layer and abstract layer), as explained latter. The relevance and benefits of a combined architecture "UDDI registry plus ontology" have been already motivated in [12] for service semantic match-making. Additional requirements in our work are due to the fact that the MAIS system is intended to support service provisioning that can be readily adapted to changes in the user context.
The main issue in MAIS is how to quickly find generic services, which we call abstract services, with the required capabilities that can be actually provided by several specific existing services, called concrete services. Abstract services are intended to shorten the way towards a variety of possible alternative concrete services that can be invoked. For this purpose, we have defined the MAIS service ontology organized into concrete services, abstract services and semantic relationships among them.
The service ontology is organized in two layers: in the *concrete layer*, concrete services are grouped into clusters according to identified semantic similarity relationships; in the *abstract layer*, abstract services are possibly related by means of semantic generalization or part-of relationships. An *association link* is maintained between each abstract service and the corresponding cluster.
During process execution, the *MAIS Registry* is directly accessed by the *Concrete Service Invocator* to find the concrete services that must be invoked. The *Platform Orchestrator*, which supports the process evolution, sends to the *Concrete Service Invocator* a sequence of service requests: the *Concrete Service Invocator* matches each request with abstract services stored into the service ontology [4]. When it finds the desired one, available concrete services belonging to the corresponding cluster are found, mapping rules are applied and services are proposed to the *Concrete Service Invocator*. At this point, context and quality requirements are checked to filter the proposed concrete services. If no concrete services in the cluster satisfy the requirements, the *Concrete Service Invocator* returns to the abstract layer and select other abstract services related to the previous one by exploiting the semantic relationships. The service ontology can be also exploited directly by users accessing the *MAIS Registry* for searching services by functionality.

The MAIS architecture also comprises a *Match Maker* component to allow for sophisticated navigation of the registry. Besides providing basic navigation functionality, it implements a set of operations for affinity evaluation for comparing the descriptions of services. The *Match Maker*,

along with the *MAIS Registry*, supports functional, non-functional, and behavioral evaluation of compatibility.

3.3 User Environment

In this section we switch temporarily from our service-centered view to a more user-centered view and present a conceptual model of a possible web-based user interface providing interaction facilities.
As this is still ongoing work, we deliberately chose a conceptual modeling approach, which allows us to ignore many of the low-level implementation-related issues without loosing expressive power. In particular, in this section we make use of *WebML*, which is a well established visual notation for the conceptual design of data-intensive Web applications and has recently been extended with new primitives also supporting the integration of Web services and thus suits out needs.

3.3.1 The Web Modeling Language

WebML is widely known for being an intuitive visual language for specifying the structure of data-intensive Web applications and the organization of contents in one or more hypertexts [9]. However, in a certain sense, it is even more than yet another specification language. Indeed, it can also be considered a full design process consisting of two main activities, which represent incremental steps towards the final application:

- **Data Design**. The WebML *Data Model* represents the basis for the overall modeling process and adopts the Entity-Relationship (ER) primitives for representing the organization of the application data. Its fundamental elements are therefore entities, attributes and relationships.
- **Hypertext Design**. The WebML *Hypertext Model* allows describing how contents, specified by means of the ER data schema, are published into the application hypertext, the so-called *site view*. Site views are structured by areas, *pages* are the actual content containers made of *content units*. They are directly associated with data entities and, by means of specific selector conditions, publish content within pages. Besides content units, *operation units* provide support for content management operations, *set* and *get units* allow accessing session variables and *entry units* model HTML input forms. Units and pages are interconnected by *links*, transporting or not parameters and describing user navigation. Figure 3 shows a graphical summary of core WebML units.

Personalization of contents and services with respect to users is achieved by modeling users and their roles as data. Personalization may occur along two different dimensions: customized contents with respect to user identy and tailored hypertext structure with respect to groups the user belongs to (e.g. guest, adiministrator). The first is based on relationships between users and content entities at data level, the latter requires designing alternative site views for each user group.
Site views may also serve the purpose of expressing alternative forms of content organizations on different devices for the purpose of *multi-channel deployment*. Each site view may cluster information and services at the granularity most

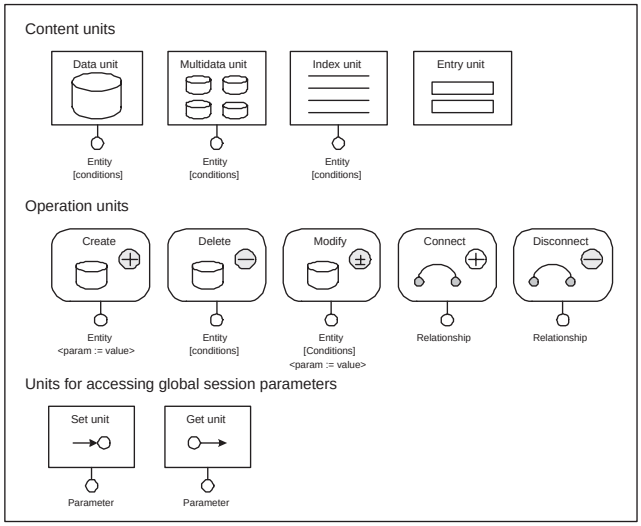

Figure 3: Summary of core WebML units.

suitable to a particular class of devices or communication protocol.
Yet WebML does not provide any delivery mechanism, nor does it depend on the particular deployment language chosen for application delivery. Its visual representation, though, is mapped on an equivalent XML-based textual representation that can be processed by *automatic code generation* tools, such as the *WebRatio Site Development Studio*.

4. RUNNING EXAMPLE

After introducing the main components of the MAIS architecture, this section exemplifies their behavior with respect to the shipping company example illustrated in Section 2.
The execution of the process specification depicted in Figure 1 is up to the *Process Orchestrator*. Its main tasks are: i) deciding when to invoke an abstract operation and ii) controlling the *link phase* of the *Concrete Service Invocator* in order to bind the choice of concrete services to the execution context.
In this example, the choice of which operation to invoke is very simple and only depends on the process specification. The orchestrator selects an abstract operation and uses the *Concrete Service Invocator* to invoke it. For instance, in the case study, the orchestrator waits until a message triggers activity *ReceiveWork*. When this happens, the orchestrator invokes the *Calculate* abstract operation, notifies the calculated plan to the driver and then waits for other incoming messages. If an *Update Route* message is notified, the orchestrator invokes the *Check* and *Replan* abstract operations and then waits again. If a *Delivery complete* message is notified, the orchestrator invokes the abstract operation *Notification* and terminates the process.
More interesting is the managing of the *link phase*. As stated in Section 2, there are various concrete services that provide operations for *Checking*, *Calculating* or *Replanning* and the selection of the suitable concrete services depends on the execution context, like, for example, the localization of the vehicle. A Driver continuously change his position and every time that the orchestrator needs to invoke an abstract

operation for checking traffic or planning the route, it must be sure that the concrete service that will perform such operation covers the geographic area where the vehicle is. This is done by forcing the *link phase* before invoking the operations *Check* or *Replan*.
The *Concrete Service Invocator* is in charge of : i) the selection of compatible concrete services and ii) the invocation of concrete operations. In our running example, the *Concrete Service Invocator* is also responsible for delivering messages between *Process Orchestrator* and *Platform Invocator*.
Initially, the *Process Orchestrator* requires that the abstract service *Route*, which contains the abstract operation *Calculate*, be linked. The *Concrete Service Invocator* searches the *MAIS Registry* for selecting concrete services that are compatible to the abstract service *Route*. This search is performed by considering constraints derived from the execution context, like the geographic position of the vehicle or the minimum GPRS bandwidth required. If it only considered geographical constraints, the *Concrete Service Invocator* would select concrete services that offer a route service that covers the geographic area in which the driver is. After the research phase, the *Concrete Service Invocator* chooses the most suitable service among selected ones. This can be done, for example, by choosing the service that offers the widest GPRS bandwidth.

After executing the *link phase*, the *Process Orchestrator* can invoke the operation *Calculate* on the linked abstract service by sending the invocation request and related abstract parameters to the *Concrete Service Invocator* . The *Concrete Service Invocator* transforms the abstract parameters into concrete parameters by means of proper wrappers and then invokes the operation on the previously selected concrete service. Returned parameters are also converted by means of the same wrapper and sent back to the orchestrator.

If the *Process Orchestrator* invokes the operation *Replan* on the previously linked abstract service *Route*, the *Concrete Service Invocator* performs such an invocation on the previously chosen concrete service or, at least, on a concrete service belonging to the same set of selected concrete services. This behavior implies that, if the *Process Orchestrator* needs to use services with a particular geographical localization, it has to perform the *link* operation every time that the vehicle changes its position.

A particular case as to the *link process* concerns the abstract service *Delivery* , which is used by the orchestrator for invoking the operation *Notification*. If we suppose that there is only one concrete service that realizes such an operation (i.e. the *ShipEveryWhere* concrete service) there is no need for the *Concrete Service Invocator* to search the *MAIS Registry* for selecting the proper concrete service. The search is avoided by the *Process Orchestrator* that performs a special *link* over the *Concrete service Invocator* in order to permanently bind the abstract service *Delivery* with the concrete service *ShipEveryWhere*.

As stated before, besides the functionality related to service invocation, the *Concrete Service Invocator* is responsible for delivering messages between the *Process Orchestrator* and the *Platform Invocator*. When the process begins, the driver must be informed about the task and subsequently about the route he has to follow. This is done by the *Process Orchestrator* that uses the functionality of the *Concrete Service Invocator* for delivering messages to the *Platform Invocator* and implicitly to the driver. The same thing is performed by the driver who uses the *Platform Invocator*, via the *User Environment*, to notify *Update Route* or *Delivery Complete* messages to the orchestrator.

The *Platform Invocator* represents the access point to the MAIS architecture. It notifies allocated tasks and related routes to the driver; this is done using an *activity list*. The *Platform Invocator* manages a list which contains all the activities (tasks, and advices, for example) assigned to drivers.

When a driver accesses the architecture via the *Platform Invocator*, he reads the assigned task, views the assigned route, and performs the delivery to the correct destination. If during the delivery process the driver decides to recalculate the route, he has to notify the decision to the architecture sending an *Update route* message via the *Platform Invocator*. The *Process Orchestrator* receives this message and reacts consequently. The same thing must be done by the driver when he completes the delivery.

Figure 4 shows a portion of the service ontology of *ShipEveryWhere*. We have four abstract services associated with the corresponding clusters of concrete services. Let us suppose that

- The *Concrete Service Invocator* receives a request of a service to replan route or to obtain traffic information from truck-A with a laptop that uses the GPRS network and requires an high bandwidth (greater than 200Kbps);
- The location scenario is the European one (context information).

The *Concrete Service Invocator* exploits the functional matching mechanism to find the abstract services *Route Planning* and *Traffic*. In the first case, it has to choose between the concrete services `PlanRoute` and `Easy Europe Travel`: For both these services the location is acceptable and both of them are provided on the GPRS network, but only the second one has an acceptable bandwidth value. So only the concrete service *Easy Europe Travel* is returned to the *Concrete Service Invocator*. The selection of a concrete service for the abstract service *Traffic* is similar. Suppose now that the same request is sent from the motorbike-D, which is equipped with a smartphone that only uses the UMTS network. The connection to the concrete services *Easy Europe Travel* and *SocietàAutostrade* is not possible, since they are only provided on GPRS networks. On the other hand, services `PlanRoute` and `TIER` are also supplied on UMTS networks and are proposed to the *Concrete Service Invocator*. Finally, let us suppose that we need a planning with cost evaluation: in this case, functional requirements concern a planning operation that returns the cost as output parameter. In our example, the *Concrete Service Invocator* uses the functional matching algorithm to obtain the abstract service *Planning with Cost*, for which, however, only the concrete service `Euro Itinerary` is acceptable, since for the service `Milan-Rome Map&guide` the location is too restrictive. So, if a GPRS network is not available, we have two solutions: the *Concrete Service Invocator* does not return any concrete services as searching result or it exploits the `is-a` relationship and presents as result the service *PlanRoute* associated with the more general abstract service `Route Planning`. This solution could be obtained also by means of a negotiation process.

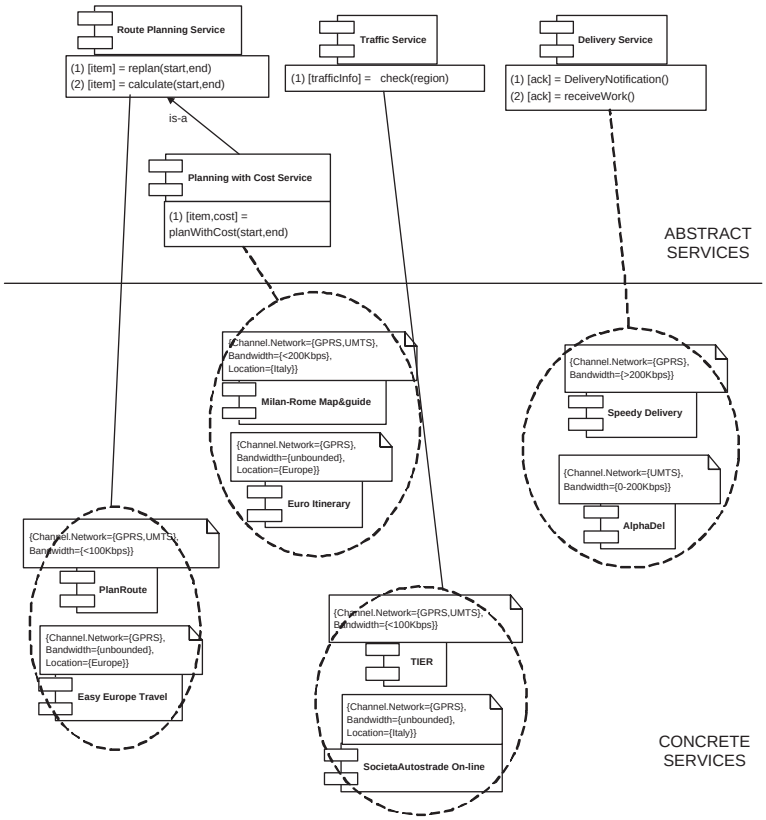

Figure 4: A portion of the service ontology for the running example.

This example shows how the service ontology can be exploited to enhance adaptive service provisioning, starting from searching abstract services with required functional capabilities, then locating groups of suitable concrete services and finally reducing the number of concrete services on the basis of context and quality requirements in an adaptive way.

4.1 Integrating Web Services and WebML

Concerning our scenario of the shipping company *ShipEveryWhere*, we require primitives capable of modeling interactions with external *Web Services*, due to the fact that the *MAIS Platform* supplies (abstract) Web services, which may be weaved into the application logic of a particular *User Environment*. [5] introduces the required functionality at an adequate level of abstraction; Figure 5 shows the graphical rendition of the units used in our example.
The depicted three operations represent just a subset of the introduced novel operations reflecting the set of WSDL message exchange patterns [10], but still enough for our purpose. The *One-way* operation serves the purpose of client-initiated messages, while the *Notify* operation stands for the inverse communication direction and thus for service-initiated messages. Finally, the *Request-Response* unit represents a synchronous operation initiated by users, with one outbound message followed by one inbound message. For further details about Web services integration into WebML please refer to [5].

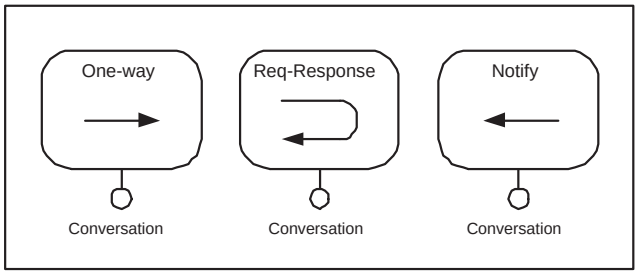

Figure 5: WebML primitives for Web services integration.

4.1.1 Data Modeling

The first step in designing the user interface regarding the van driver in our example consists of modeling the application data. Starting from the default WebML sub-schema, required for user management and personalization, three entities (*Van, Package, Route*) model the specific application data. The execution of service-related operations causes implicit update of data. In particular, the operations *GetPackageList* and *GetRoute/NewRoute* affect the entities *Package* and *Route* respectively.

4.1.2 Hypertext Modeling

Figure 7, finally, shows the arrangement of a possible hypertext built upon the specified data model and gives an idea of the complexity masking power of the *MAIS Plat-*

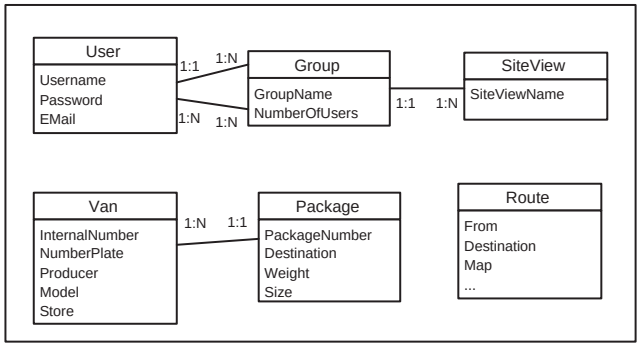

Figure 6: Data model for integrating the *ShipEvery-Where* service.

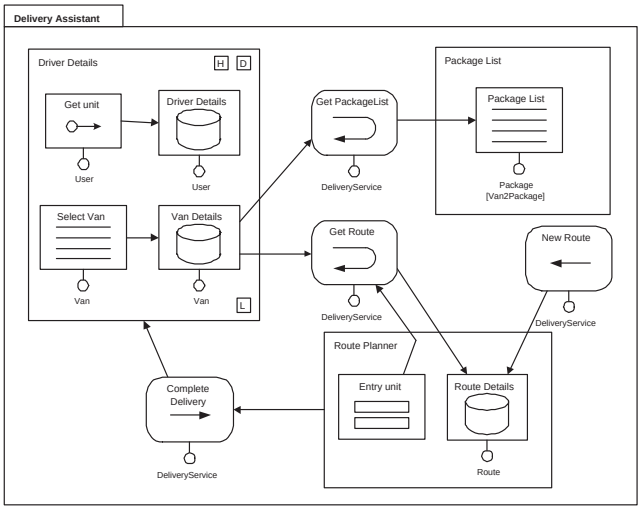

Figure 7: Hypertext schema of user interface.

form. Only operations exposed by means of *abstract MAIS services* are known at the *User Environment* level, while all the process orchestration details occur in a completely transparent manner. The names of used operations are referred to the user environment and are directly mapped onto the operations of the process as described in Figure 1: Grey lines correspond to user-oriented invocations.

The hypertext schema describes the PDA user interface of the van driver. After a successful login process (not modeled here), the page *Driver Details* shows the relative user details and provides a list of free vans. The driver chooses one of the vans and, based on the chosen van, he can either get the list of the loaded packages or get the delivery route for the freight. Both these operations are performed by means of calls of the delivery Web service. While visiting the *Route Planner* page, the route can even change automatically due to changing traffic conditions and notified by means of the notify unit *New Route*. At the other hand, also the driver himself can manually invoke the *Get Route* operation by means of the entry unit. Once the packages have been delivered, the *Complete Delivery* operation communicates the successful delivery back to the control center.

5. RELATED WORK

The semantic description of services is very important in dynamic contexts where different services can offer, completely or partially, the requested features. The use of a registry with publish and subscribe capabilities is the usual way to allow a dynamic search of services. The de-facto standard for registries is UDDI and nowadays all the semantic match-makers must be UDDI-compliant. The description of interfaces by means of WSDL is UDDI-compliant, but it is not enough to perform useful semantic researches in a service registry. On the other hand, the use of rich descriptions, followed by the OWL-S coalition [1], can raise problems like the compliancy with UDDI and the delay associated with searches.

A compromise is described in [12, 22], where the authors propose a semantic description of services and a match-maker able to browse a UDDI-compliant registry. Our approach follows this compromise since the semantic description is used to improve the degree of freedom in the design of the business process, but search performances are still acceptable.

Another important issue as to service provisioning concerns the definition of languages for Quality of Service (QoS) descriptions. QoS has been the topic of several research and standardization efforts across different communities [18, 21, 23]. In [15], authors propose a multilayer model to evaluate quality of services in a dynamically evolving environment.

The adaptivity to the context is a fundamental issue of modern frameworks for provisioning of services. The adaptation process can involve or not the user. According to the degree of user interaction, we can identify three different levels.

At the lower level, adaptivity is focused on the middleware for service provisioning [7, 16]. In this perspective the nature of the application is weakly considered and often the user does not know or interact with the adaptation process. The middle level is related to adaptivity issues on the business logic. Here, applications can react to events forwarded by the lower levels and modify their business logic in order to adapt their behavior with respect to users. Several systems and approaches have been proposed to extend traditional workflow management system technology to adaptive, Internet-based scenarios: CrossFlow [11] , WISE [14], MENTOR-LITE [19]. e-FLOW [8] is one of the first research prototypes addressing the issues of specifying, enacting, and monitoring composite services; other proposals include SELFSERV [3] in which services can be composed and executed in a decentralized way and The Dysco project [17] that faces the issue of automatic composition.

The top level involves aspects related to user environments [6]. Applications modify their user interfaces according to the client execution context. Automatic transcoding tools, like WebML [9], are very important in the automatic generation of multi-channel access systems.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a novel approach for the provisioning of complex abstract services. The decoupling of abstract description of services and their actual implementations is strongly exploited by the MAIS architecture and it was designed with this purpose in mind. Our service ontology is defined by looking a compromise between the richness of the description and its real usability. The defini-

tion of QoS dimensions become the fundamental parameter for the selection of the best service.
The possibility of dynamic search is already a kind of adaptivity. Moreover to increase the flexibility our framework, we can provide simple services that have to be orchestrated by the end user or the architecture can hide all details and present only a value-added (fully orchestrated) service.
The adaptivity is also addressed by using a reflective architecture, which is able to know and, in some case manage, the parameters of the distribution channel.
Even if exiting languages give many opportunities, it is necessary to augment some of them. We are now formalizing these extended languages. The next step will be the implementation and deployment of the MAIS framework in some special-purpose settings.

Acknowledgments

This work is partially funded by the Italian MURST-FIRB MAIS Project (Multi-channel Adaptive Information Systems).

7. REFERENCES

[1] A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. M. D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, and K. Sycara. Daml-s: Web service description for the semantic web. In *In Proc. of International Semantic Web Conference (ISWC 2002)*, Chia, Italy.

[2] V. D. Antonellis, M. Melchiori, B. Pernici, and P. Plebani. A methodology for e-service substitutability in a virtual district environment. In *Conference on Advanced Information System Engineering (CAISE 2003), Klagenfurt-Velden, Austria, June 16-20, 2003*.

[3] B. Benatallah, Q. Sheng, and M. Dumas. The self-serv environment for web services composition. *IEEE Internet Computing*, 7(1), 2003.

[4] D. Bianchini, V. D. Antonellis, and M. Melchiori. An ontology-based method for classifying and searching *e*-Services. In *Proc. Forum of First Int. Conf. on Service Oriented Computing (ICSOC 2003)*, Trento, Italy, December 15-18 2003.

[5] M. Brambilla, S. Ceri, S. Comai, P. Fraternali, and I. Manolescu. Model-driven specification of web services composition and integration with data-intensive web applications, July 2002. IEEE Bulletin of Data Engineering.

[6] P. Brusilovky. Adaptive hypermedia. *User Modeling and User Adapted Interaction*, 11(1-2):87–100, 2001.

[7] L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-aware reflective middleware system for mobile applications. *IEEE Transactions on Software Engineering*, 29(10):929–945, 2003.

[8] F. Casati and M. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 6(3), 2001.

[9] S. Ceri, P. Fraternali, B. Bongio, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. 2002.

[10] M. G. et al. *Web Services Description Language (WSDL) Version 2.0 Part 2: Message Patterns*. W3C, http://www.w3.org/TR/wsdl20-patterns/, November 2003. W3C Working Draft.

[11] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. Crossflow: Cross-organizational workflow management in dynamic virtual enterprises. *International. Journal of Computer Systems Science & Engineering*, 15(5), 2000.

[12] T. Kawamura, J. D. Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Preliminary report of public experiment of semantic service matchmaker with UDDI business registry. In *In Proc. of International Conference on Service oriented Computing ICSOC*, volume 2910/2003, pages 208–224, Trento, italy, 2003. Lecture Notes in Computer Science Springer-Verlag Heidelberg.

[13] J. Krogstie, K. Lyytinen, A. L. Opdahl, B. Pernici, K. Siau, and K. Smolander. Research areas and challenges for mobile information systems. *Int. J. Mobile Communication*, in press.

[14] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The WISE approach to electronic commerce. *International Journal of Computer Systems Science & Engineering*, 15(5), 2000.

[15] C. Marchetti, B. Pernici, and P. Plebani. A quality model for multichannel adaptive information systems. In *In proc. of WWW04 Conf., alt. track on Web Services*, 2004.

[16] N. Parlavantzas, G. Coulson, and G. Blair. A resource adaptation framework for reflective middleware. In *In Proc. of International Middleware Conference, Workshop*, pages 163–168, Rio de Janeiro, Brazil, 2003.

[17] G. Piccinelli and L. Mokrushin. Dynamic e-service composition in dysco. In *In Proc. of Int. Workshop on Distributed Dynamic Multiservice Architecture, at ICDCS*, Phoenix, Arizona, USA, 2001.

[18] S. Ran. A model for web services discovery with qos. In *ACM SIGecom Exchanges*, volume 4(1), 2003.

[19] G. Shegalov, M. Gillmann, and G. Weikum. XML-enabled workflow management for e-services across heterogeneous platforms. *VLDB Journal*, 10(1), 2001.

[20] The MAIS Project Team. The MAIS Project. In *International Conf. on Web Information Systems Engineering, Roma, Italy*, Dec. 2004.

[21] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *In Procóf Conference on World Wide Web*. ACM Press, 2003.

[22] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Tqos-aware middleware for web services composition. *IEEE Trans. on Software Engineering*, 30(5), May 2004.

[23] J. Zinky, D. Bakken, and R. Schantz. Architectural support for quality of service for corba objects. In *Theory and Practice of Object Systems*, volume 3(1), 1997.