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 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 between the general characteristics of services and the characteristics associated with service invocation. In many cases, 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 services in a mobile and interactive environment and we focused 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 process level, during the selection of a concrete service, and also at end user level. Selection is based on suitable ontologies and considers 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 simple process for a shipping company.

Keywords

Service-oriented computing, Service discovery, service selection

1. INTRODUCTION

The emerging paradigm of service-oriented computing supports the creation of applications by composing services selected among a variety of available services with different characteristics. Services may be invoked directly by the application in which they are used. Essential to this paradigm is the description of services using a standardized approach; in the literature, several proposals of service description languages have been made, such as WSDL, of service ontologies, such as in AgFlow [25], of semantic web services [1], separating general characteristics of services from the characteristics related to service invocation. In the above mentioned approaches, service selection is generally static, assuming matching techniques to retrieve the most appropriate services. In VISPO [2], authors introduce 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 concrete service. During selection and execution, the availability of the selected process is evaluated and mechanisms for substituting concrete services whenever they are not available are provided.

However, in both VISPO and AgFlow, where service unavailability is considered at runtime, 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 highly variable environment in terms of both the architecture and its components. In such environments, for example in the case of mobile information systems [15], services invocation may vary depending on their availability over the network, on parameters of devices on which they are invoked and on the characteristics of the networking infrastructure. In addition, services might be used in the process several times and their execution environment might change over time.

The goal of this paper is to propose the MAIS architecture and its mechanisms for designing and executing complex services exploiting adaptivity. We propose adaptivity at different levels: at the process level, at the level of selection of a concrete service for a given abstract service and at the level of interface between users and the platform. We support service selection with an enhanced UDDI registry, storing descriptions of abstract and concrete services and including information about quality parameters on the provider side. The proposal has been developed in the MAIS (Multichannel Adaptive Information Systems) Project [23].

The rest of this paper is organized as follows: Section 2 introduces the scenario for our running example, a mobile information system for a shipping company in order 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 services selection and invocation. It also introduces the basic services ontology of the MAIS Registry. Section 4 exemplifies the behavior of the functional architecture with respect to the running example and discusses a mechanism for decoupling service invocation from the design of a user environment in terms of its interaction with the system, based on an extension of the WebML

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

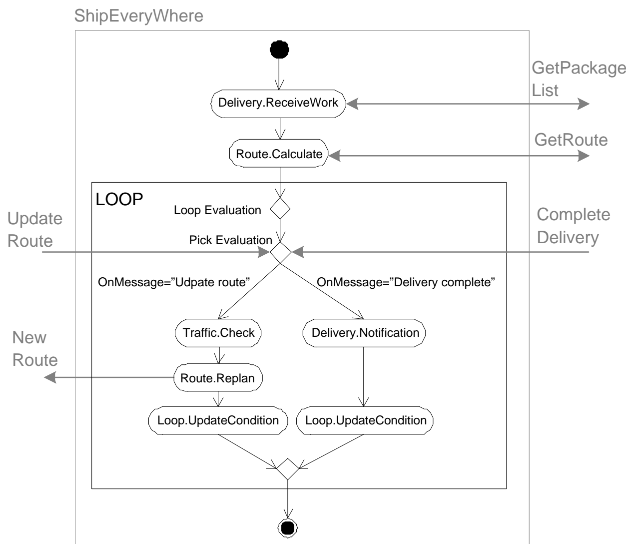

Figure 1: Workflow of shipping assignment and execution phases

model [10]. Section 5 discusses our proposal in relation to the state of the art. In Section 6 conclusions are reported and future work is anticipated.

2. THE SHIPPING COMPANY

This section presents the scenario of 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 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 the 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 workflow model of the process that oversees all the phases and is inspired by basic BPEL4WS [11] primitives. Notice that we use a dot notation to name activities: the first part identifies the service, while the second part specifies the operation.

The process starts with the assignment of the task to the vehicle that carries the item (service *Delivery*, operation *ReceiveWork*). The *ShipEveryWhere* control center, through an appropriate user interface, assigns the item and related delivery information to the selected driver. According to the destination and dimension of the item, the vehicle can be a truck, for destinations longer than 200 kilometers, a van, for destinations 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 motorbike drivers use a smart-phone. Vehicles are equipped with different devices because of the different uses and the available room on board. This choice, however, implies that the enactment of the process varies and that it 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 conditions and consequently replan the route. The *UpdateRoute* operation 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 *UpdateRoute* 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 placed (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, because the bandwidth is too low, he can 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 functional architecture, its components and the relationships among them.

Before doing this, we must set the jargon used in the paper and clarify that we distinguish between:

- *abstract services*, which are services that cannot be invoked directly and for which only abstract aspects, like functional interface and QoS levels, are described; implementation details are omitted; the functional description is expressed using the abstract part of the WSDL specification, in terms of the operations that the service performs, the input values it requires for the execution and the output values it produces after execution; constraints on input and output values can be specified; 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 (see [18]);

- *concrete services*, which are services directly invocable, which inherit the abstract functional interface and the QoS levels of an abstract service; a concrete service can extend the functional interface and QoS description with implementation specific details (i.e., access protocols, QoS values); the concrete services are clustered on the basis of their functional similarity (see [4]), evaluated on the WSDL abstract interfaces; abstract services are associated to the clusters of concrete services

and their interfaces are representative of the interfaces of the concrete services in the clusters; in particular, the set of operations in the interface of the abstract service are only those common to all the concrete services in the cluster; the designer can possibly force further capabilities that are considered to be relevant for the cluster, for example, because they are present in most concrete services of the cluster; in all cases, 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; furthermore, in our framework concrete services are distinguished into two subcategories:

  - *simple concrete services*, which are concrete services that do not use the orchestration and substitution functionalities of the MAIS architecture; all these operations are made by the provider and are hidden for our framework and for users;
  - *complex concrete services*, which are concrete services containing an abstract process definition which will be instantiated and executed using the orchestration and substitution functions offered by our framework.

  Contextual information is given in terms of conditions under which the service is provided and are used to further refine concrete services presented to the end users; in particular, it refers to the *Channel* used for service provisioning, the *Location* where it is used and the *Time* at which it is executed;

- *MAIS services*, which are generic services offered by the MAIS framework to end users; this definition masks the framework complexity, hiding service classification to end users, who work only with MAIS services while the management of the different kind of services is performed by the framework.

There are two kinds of actors which interact with the MAIS architecture:

- *designers*, who define processes contained into the complex concrete services; designers browse the registry to search for MAIS services, select them, and compose processes with the selected services;
- *end users*, who invoke the MAIS services; end users interact with the platform for searching and invoking services.

3.1 MAIS architecture

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

The *Platform Invocator* represents the point of contact between the *User Environment* and the MAIS architecture and hides the complexity of the architecture. Its interface exports a set of operations, which allow interacting programmatically with the architecture, performing operations like: i) searching published services in the *MAIS Registry* ii) executing the chosen services and iii) managing the interaction with the *User Environment* during the execution of a complex service. 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 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., devices, available communication protocols, user profile). This module is realized using *WebML* (see Appendix), 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 our needs.

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

Once end users select and invoke a service using the *User Environment*, 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 service operations.

The *Process Orchestrator* manages the state of the process and, step by step, interacts with the *Concrete Service Invocator* for invoking each operation specified in the definition of the workflow. 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.

The *Concrete Service Invocator*, which is in charge of managing the invocation of services, can:

- Start the invocation of concrete services. When the *Platform Invocator* asks for a service invocation, the *Concrete Service Invocator* invokes the correct concrete service after the interaction with the *MAIS Registry.*

- Invoke abstract operations. This is a sophisticated functionality used by the *Process Orchestrator* for invoking abstract operations. An abstract service cannot be invoked and we need to select concrete services. 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 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.*

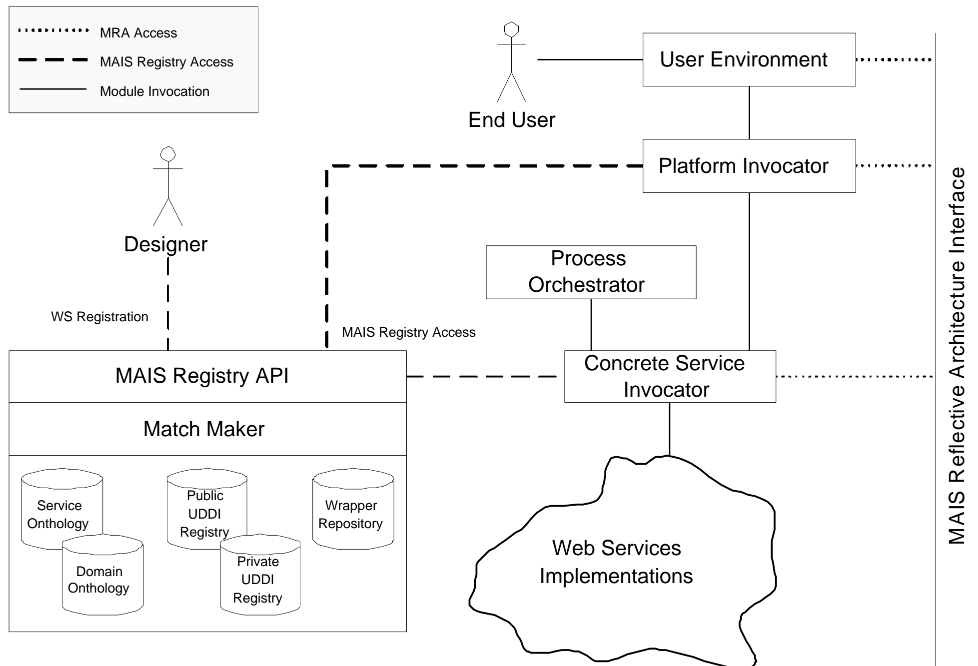

Figure 2: MAIS functional architecture

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

The last module of our architecture is the *MAIS Registry*. This module contains suitable descriptions of all published services, along with other auxiliary information.

The MAIS registry 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 later. The relevance and benefits of an architecture, which combines UDDI registries with ontologies, have been already motivated in [14] for semantic service matchmaking.

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 related to each other by means of semantic generalization and/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. Given an abstract service, the *MAIS Registry* searches the available concrete services associated to the abstract one, applies mapping rules and proposes services to the *Concrete Service Invocator*. At this point, context and quality requirements are checked to filter the proposed concrete services.

The MAIS architecture also comprises a *Match Maker* component, which provides the functionalities for publishing and retrieval services on which the *MAIS Registry API* is based. During the publication phase, the *Match Maker* analyzes the published service description, compares it with service descriptions that already exist in the *MAIS Registry* using a set of techniques for affinity evaluation and updates the contained UDDI registry and the ontologies. Furthermore, the *Match Maker* acts as search engine for browsing the registry and the ontologies, in order to retrieve all the concrete services compatible with an abstract one.

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* 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 shipping company example, the orchestrator waits until a message triggers the activity *ReceiveWork*. When this happens, the orchestrator invokes the abstract operation *Calculate*, notifies the calculated plan to the driver and then waits for other incoming messages. If an *UpdateRoute* message is notified, the orchestrator invokes the abstract operations *Check* and *Replan* and then waits again. If a *CompleteDelivery* message is notified, the orchestrator invokes the abstract operation *Notification* and terminates the process.

More interesting is the managing of the *link phase*. 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 position of the vehicle. A driver continuously changes 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 performs 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*.

Initially, the *Process Orchestrator* requires the abstract service *Route* to be linked, which contains the abstract operation *Calculate*. 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. For example, if we only consider 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 search 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 *Process 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 concrete service already selected. If such service is unavailable, a concrete service belonging to the same set of selected concrete services must be chosen. This behavior implies that, if the *Process Orchestrator* needs to use services with a particular geographical position, it has to perform the *link* operation every time that the vehicle changes its position.

A particular case of 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* to permanently bind the abstract service *Delivery* to 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 *UpdateRoute* or *CompleteDelivery* 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 *User Environment*, 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 control center by sending an *UpdateRoute* message via the *User Environment*. The *Process Orchestrator* receives this message and reacts consequently. The same thing must be done by the driver when he completes the delivery.

Moving to the MAIS Registry, Figure 3 shows a portion of the service ontology of *ShipEveryWhere*. We have four abstract services associated with the corresponding clusters of concrete services. We 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 a high bandwidth (greater than 100Kbps);
- 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 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 concrete services as 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 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

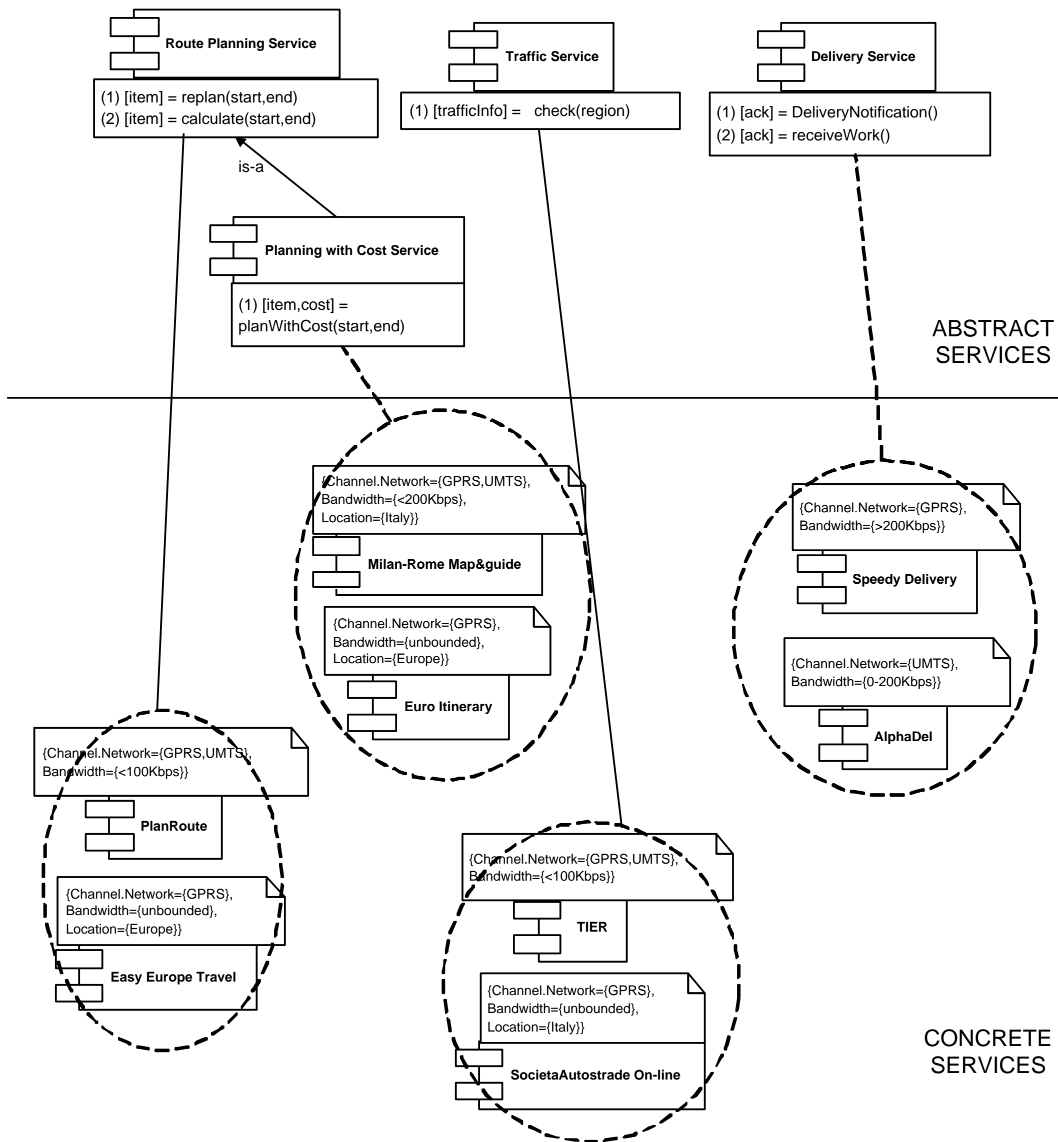

Figure 3: A portion of the service ontology for the running example.

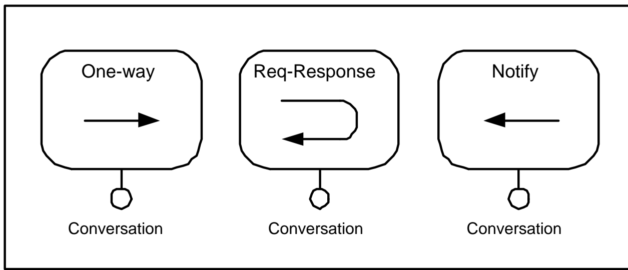

Figure 4: WebML primitives for Web services integration

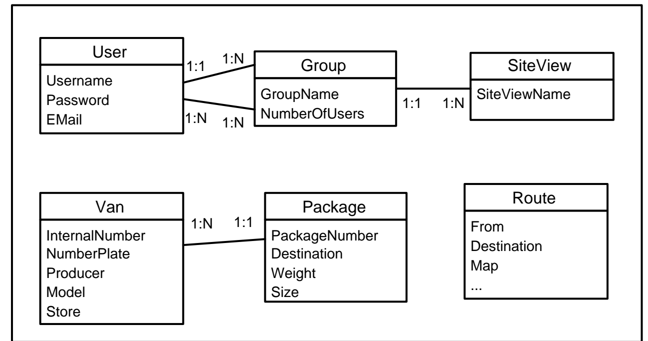

Figure 5: Data model for integrating the *ShipEveryWhere* service

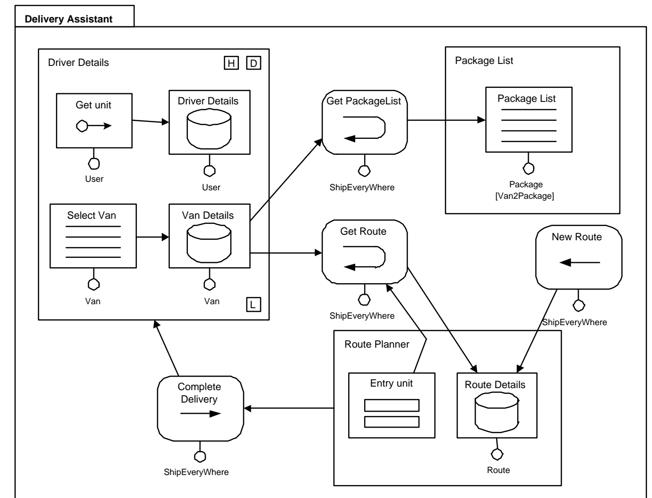

Figure 6: Hypertext schema of user interface.

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 architecture supplies (abstract) Web services, which may be weaved into the application logic of a particular *User Environment*. [5] introduces the required functionalities at an adequate level of abstraction; Figure 4 shows the graphical rendition of the units used in our example. Appendix A shows the WebML constructs used for this purpose.

The depicted three operations represent just a subset of the introduced novel operations reflecting the set of WSDL message exchange patterns [13], 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, 6, 17].

4.1.1 Data Modeling

The first step in designing the user interface regarding the van driver 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 6, 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 Platform*. 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: incoming or outgoing links with respect to the box called *ShipEveryWhere*, which represents the overall delivery service, 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 *NewRoute*. At the other hand, also the driver himself may send manually an *UpdateRoute* message and wait for the asynchronous answer *New Route*. Once the packages have been delivered, the *CompleteDelivery* 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 that publishes and subscribes capabilities is the usual way to allow a dynamic search of services. The de-facto stan-

dard 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 search 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 [14, 25], 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 aspect of 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 [21, 24, 26]. In [18], 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 [8, 19]. 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 [12] , WISE [16], MENTOR-LITE [22]. e-FLOW [9] 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 [20] that faces the issue of automatic composition.

The top level involves aspects related to user environments [7]. Applications modify their user interfaces according to the client execution context. Automatic transcoding tools, like WebML [10], 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 definition 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 of 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 channels.

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. Martinand 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, 2002.

[2] V. De Antonellis, M. Melchiori, B. Pernici, and P. Plebani. A methodology for e-Service substitutability in a virtual district environment. In *Proc. of 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, volume 2681 of *Lecture Notes in Computer Science*, pages 552–567, Klagenfurt, Austria, June 16th-20th 2003. Springer.

[3] B. Benatallah, Q. Sheng, and M. Dumas. The Self-Serv Environment for Web Services Composition. *IEEE Internet Computing*, 7(1):40–48, 2003.

[4] D. Bianchini, V. De 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 15th-18th 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. *Bulletin of the Technical Committee on Data Engineering*, 25(4), December 2002.

[6] M. Brambilla, S. Ceri, S. Comai, P. Fraternali, and I. Manolescu. Model-driven development of web services and hypertext applications, December 2003. SCI2003, Orlando, Florida.

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

[8] 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.

[9] F. Casati and M. Shan. Dynamic and Adaptive Composition of e-Services. *Information Systems*, 26(3):143–163, May 2001.

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

[11] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business*

Process Execution Language for Web Services, version 1.0, July 2002.

[12] 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):277–290, 2000.

[13] Martin Gudgin, Amy Lewis, and Jeffrey Schlimmer. Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extension, August, 3rd 2004.

[14] T. Kawamura, J.A. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Preliminary Report of Public Experiment of Semantic Service Matchmaker with UDDI Business Registry. In *Proc. of First Int. Conf. on Service Oriented Computing (ICSOC 2003)*, volume 2910, pages 208–224, Trento, Italy, December 15th-18th 2003. Lecture Notes in Computer Science Springer-Verlag.

[15] J. Krogstie, K. Lyytinen, A. L. Opdahl, B. Pernici, K. Siau, and K. Smolander. Research areas and challenges for mobile information systems. *International Journal of Mobile Communication (IJMC). Special issue on Modeling Mobile Information Systems: Conceptual and Methodological Issues*, 2(3), 2004.

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

[17] I. Manolescu, S. Ceri, M. Brambilla, P. Fraternali, and S. Comai. Exploring the combined potential of web sites and web services. poster at WWW03, Budapest, Hungary.

[18] C. Marchetti, B. Pernici, and P. Plebani. A Quality Model for Multichannel Adaptive Information Systems. In *Alternate Tracks Proceedings of 13th International World Wide Web Conference (WWW2004), ACM Press*, pages 48–54, New York City, NY, USA, May 17th-22th 2004.

[19] N. Parlavantzas, G. Coulson, and G.S. Blair. A Resource Adaptation Framework For Reflective Middleware. In *Proc. of 2nd Int. Workshop on Reflective and Adaptive Middleware (located with ACM/IFIP/USENIX Middleware 2003)*, pages 163–168, Rio de Janeiro, Brazil, June 2003.

[20] 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.

[21] Shuping Ran. A Model for Web Services Discovery with QoS. In *ACM SIGecom Exchange*, volume 4, pages 1–10, ACM Press, New York, NY, USA, 2003.

[22] G. Shegalov, M. Gillmann, and G. Weikum. XML-enabled Workflow Management for e-Services across Heterogeneous Platforms. *VLDB Journal: Very Large Data Bases*, 10(1):91–103, 2001.

[23] The MAIS Project Team. The MAIS Project. In *Proc. of 4th International Conf. on Web Information Systems Engineering*, Rome, Italy, December 2004.

[24] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *In Proc. of Conference on World Wide Web*, pages 411–421. ACM Press, 2003.

[25] L. Zeng, B. Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. QoS-Aware middleware for web services composition. *IEEE Trans. on Software Engineering*, 30(5):311–327, May 2004.

[26] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1):1–20, 1997.

APPENDIX

A. 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 [10]. 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:

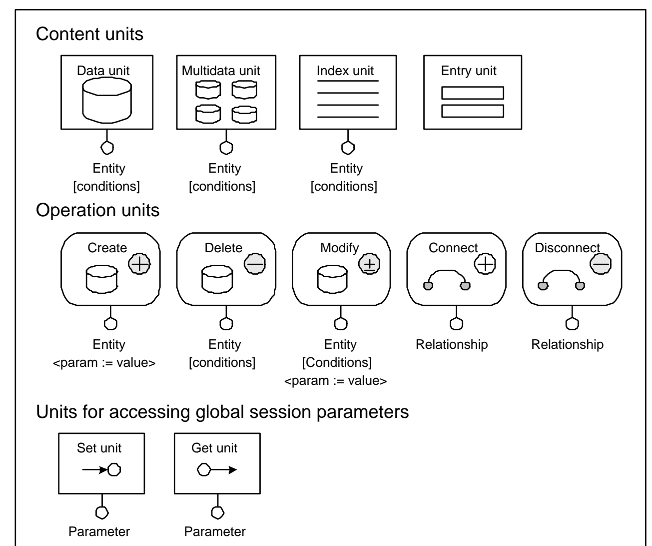

Figure 7: Summary of core WebML units.

- **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 views*. Site views are structured by *areas* and *pages*, that 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 7 shows a graphical summary of core WebML units.

Personalization of contents and services is achieved by modeling users and their roles as data. Personalization may occur along two different dimensions: customized contents with respect to user identity and tailored hypertext structure with respect to groups the user belongs to (e.g., guest, adiministrator and so on). 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 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*.