Putting Together Web Services and Compositional Software Agents*

Mercedes Amor, Lidia Fuentes, and José María Troya

Depto. Lenguajes y Ciencias de la Computación

Universidad de Málaga

Campus de Teatinos s/n 29071 Málaga

{pinilla, lff, troya}@lcc.uma.es

Abstract. Web services are the newest trend in information technology, being considered the most used alternative for building distributed open systems. Although currently Web services involve a single client-server access, the market is demanding cooperative Web services to provide a global solution. Recently software agents appear as a good option that can cope with the control of Web services composition, obtaining an integral solution. This paper presents an approach to integrate Web services and software agent technologies. The basis of our approach is the use of the component technology for the development of adaptive software agents. Our compositional software agent performs automated software composition based on the flexibility provided by the component orientation, which makes possible to plug Web services into the agent functionality and compose them during the agent interaction.

1 Introduction

Computer industry increasingly focuses on Web services as an alternative to build distributed open systems above Internet. Any Web-accessible program can be considered one of this kind of services. Web services are the newest trend in information technology, because they are XML-based, can operate through firewalls, are lightweight, and are supported by all software companies. Web services are accessible through standardized XML messaging, with no care about how each Web service is implemented. Web services use and support an important set of new standards proposals, such as XML Schema [1], SOAP [2] (Simple Object Access Protocol), UDDI [3] (Universal Description, Discovery and Integration), and WSDL [4] (Web Services Definition Language). Besides these standards, Web service developers and users need to broadly agree on the semantics of specific domains. The Resource Description Framework (RDF) [5], DARPA Agent Modeling Language (DAML+OIL) [6], and ontologies in general support this agreement.

* This work has been partially supported by CYTED project VII.18 ("WEST: Web Oriented Software technology").

But Web services technology also presents some shortcomings [7]: a Web service is not so adaptable, and it is unable to benefit from new capabilities of the environment to provide improved services; commonly Web services are not designed to use ontologies, and if the service provider and the client happen to use different ontologies the result of the service invocation could not be satisfactory; Web services are passive until they are invoked, and they are not able to provide alerts and updates when information becomes available; and finally, the most important drawback is that currently there is no standard to support composing functionalities among Web services. Concluding, although currently Web services involve a single client-server access, the market is demanding cooperative Web services to provide integral solutions.

Agent technology is now being used to interact with Web services on behalf of a customer, trying to overcome some of these Web services drawbacks. For instance, software agents can send alerts and updates when Web service information becomes available, and they can manage the correct use of ontologies. Above all, software agents are able to integrate different Web services into a unique solution. Software agents make use of a Web service by accessing to its semantic description, especially in terms of an ontology specified in a well-known language. That is, a software agent needs a computer-interpretable description of the service, for example in DAML-S [8].

Some efforts have recently started to integrate the software agent and the Web service communities. The FIPA services group has initiated a specification that enables FIPA agents to use Web services infrastructure (e.g. the message transport service), and propose to extend the Web services model with the benefits of agent technology [9]. Similarly, AgentCities has established a working group to integrate Web services architecture into their AgentCities framework [10]. This working group is examining the complexity of service descriptions and the dynamic composition of Web services in open environments. On the contrary, however, the Web service community does not seem to be interested in integrating agent technology into its infrastructure.

We propose to integrate Web service and software agent technologies through our component-based architecture for developing software agents [11][12], where agent behaviour is provided by different plug-ins components. In order to improve agent modularisation we also apply the separation of concerns technology [13]. Concretely, functionality and coordination issues are separated internally in different entities inside the architecture. The composition between agent internal components is performed at runtime, allowing the reconfiguration and adaptation of agent behaviour to support new interactions, for instance with Web services.

The benefit of component-oriented systems lies on their increased flexibility: a system built from components should be easier to recompose to address new requirements [14]. A compositional design may allow software agents to be dynamically adapted to support new negotiation protocols and new functionality, by plugging new software components.

Since agent functionality is provided by in-house and/or Commercial-Off-The-Shelf (COTS) components, a Web service can be considered as a special kind of a software component. Likewise components, Web services represent black-box functionality, and can be reused without worrying about how the service is implemented, being really loosely coupled. While a software component provides its functionality through well-defined interfaces, a Web service is offered through a XML standard service

description, which provides all the necessary details to interact with the service. A Web service interface is defined strictly in terms of the messages the Web service accepts and generates as response, the message format (e.g. inside SOAP) and the transport protocol (normally HTTP).

In addition, we propose a single interface description of all the components, without taking care of how a component is accessed or implemented. In order to make possible components and agents interoperability it is needed a syntactic and a semantic agreement of component provided interfaces. We use a subset of DAML-S, an ontology for services description, as our component interface description language.

2 Software Agent Compositional Architecture

In this section we describe the core of our approach, a compositional architecture for software agents. Software agents are systems that perform actions or tasks autonomously to reach a set of goals. Software agents are able to perform complex interactions including negotiation and collaboration. The main characteristic of our model is that encapsulates agent functionality in software components. Modelling agent functionality as components we found out that a software agent contains components for data storing (e.g. to store its internal goals, facts, etc.), for providing application domain functionality (e.g. buy, bid, search actions), and components that coordinate the agent internal and external execution.

To improve agent functional decomposition we apply the principle of separation of concerns, that is, we model the coordination issue separately in a new entity called connector. We mean by coordination, the flow control between agent inner components and/or external agents. Thus connectors may model negotiation protocols. We have already successfully applied the Aspect Oriented Programming (AOP) to separate the coordination issue in some previous works [15][16], so we endow the connector concept to agents, mainly to increase their adaptability. In this case, we improve the agent dynamic adaptation to a new interaction protocols through the runtime composition of components and connectors inside the software agent. Fig. 1 shows the UML class diagram of the proposed compositional architecture of a software agent based on components and connectors. We use UML stereotypes for modelling the principal entities of our model, which are Component, Connector, Mediator and Interface. To simplify Fig. 1, we represent data and behaviour classes that model components as a subsystem, and connector class as a coordination subsystem ([12] contains a more detailed description of the UML class diagram). Connectors coordinate the different interactions the agent is involved. An agent can participate in more than one conversation simultaneously, and one protocol connector controls each conversation. Connectors coordinate dialogues according to a specific interaction protocol (e.g. English Auction Negotiation Protocol). These connectors differ only in the coordinated protocol, given by a template filled by the protocol specific rules, transitions and constraints.

Components encapsulate data and behaviour. Some behavioural components are always presented in the architecture providing generic agent functionality, such as to

send a message, or to store general data. Domain specific behaviour is provided also by components. Specific components offer functionality to perform, for example, e-commerce tasks, such as to generate a bid. Components will be plugged into agent functionality on demand, and are changeable over the lifetime of the agent. Hence, domain specific behaviour can also be provided by Web services or COTS components.

Fig. 1. Compositional Architecture for Software Agents.

Interface components manage the interactions of the agent. The *AgentExternalCommunication* (AEC) component processes the input messages and sends replays to an agent platform for delivery. Incoming messages are first processed by this component, and it may discard messages that are not syntactically well formed. The component *AgentInterface* (AI) contains the public agent interface, which is an extension of the traditional IDLs of software components. For instance, AI includes, in addition to the provided services descriptions, a list of public negotiation protocols, etc., (the agent interface description is beyond the scope of this paper). Concerning this paper, this component mainly stores the templates of the supported protocols, and the ontologies committed by the plugged components.

The *AgentCompositionalCore* (ACC) mediator component mainly performs the dynamic composition of components and connectors. The ACC receives input messages from the AEC component, dispatching them to the appropriate conversation according to the conversation identifier contained in the message. In addition, this component launches the connector that encapsulates a specific protocol as a new interaction is initiated. The mediator also maintains the relevant information about components that provide agent functionality. This information is represented in the architecture by the class *ComponentInfo*. Each component is registered within the agent architecture with a kind of role name, which is used to identify the component for late binding purposes. Since the role a component plays inside an agent is determined by the ontology it commits, we use the ontology name for identifying components. In addition it is

necessary to indicate the location of the component. If the component is an internal instance, the *localAddress* field contains the local reference of the component. In other case, that is, the component is an external COTS component or a Web service, the *remoteAddress* field will contain the URL to access to component services.

Dynamic composition is used for the late binding between coordination connectors and the agent behaviour provided by components. A relevant characteristic of our model is that components and connectors have no direct references among them (they are referenced by the ontology name), which increase their independency. Dynamic composition also allows plugging a new component implementation into the agent dynamically, or upgrading this compositional agent without replacing the agent. This feature makes feasible to plug Web services and/or COTS components easily. Even the agent configuration can change to replace a registered service provider (a component, Web service, or a COTS component) for providing a service more efficient, with a lower cost, or valuable. For example, an agent can test different Web service providers that offer the same service, and use the one with the best response time.

3 Service Ontology

For composing coordination and agent behaviour at runtime the agent has to match up connector required services with the offered services of the registered component. In order to do that, components must provide some knowledge mainly about the services they provide. Web services, COTS components, and in general software components provide their functionality through well-defined interfaces. The way the connector requests the services is through the Mediator component, which knows registered components and their interfaces, since the connector must be unaware of the type of the plugged components. Consequently we need to use a common access method independent of the implementation language and location of the component, unifying the interface description of different software components that provide agent functionality. A simple solution could be to match connector service request by using just the syntactic description of component public methods, i.e. their signatures. However, it is required to share a common agreement about the meaning of the terms used in the interfaces, i.e. a common vocabulary. Since most IDLs only include a syntactic definition of the offered services, and any semantic information, we will enrich component interfaces with a semantic description of the services.

We consider the use of ontologies for describing components public interfaces. Ontologies are a powerful tool to enable knowledge sharing, and reach semantic interoperability. Actual ontologies in use focus on service advertising and discovery [17]. We propose to use DAML-S to describe components provided and even required functionality. DAML-S is a DAML+OIL ontology for describing the properties and capabilities of Web services, and it is as part of DARPA Agent Markup Language program. DAML-S complements low-level descriptions of Web services describing what a service can do, and not just how it does it. DAML-S is a Web markup language that provides a declarative, computer-interpretable API for executing function calls. The description of a service using DAML-S comprises three different views of the service:

what the service requires and provides the user, or agent is described in the service profile; how the service works is given in the process model; and, how to use the service is described in the grounding.

In our model, if an agent commits a given ontology means that there is a local component that implements the functionality required by the ontology, or knows an external component (e.g. a Web service) that provides it.

Now, we will walk through a small example to show how to define and describe a service using DAML-S. Consider the case of a Web service that provides an account transfer service, accessible through a WSDL description (WSDL is an XML-based language used to define Web services and describe how to access them). Apart from grounding we need to provide the DAML-S profile and process model of the Web service to describe a service in DAML-S. The profile and the process ontologies define classes and properties that must be specialized to describe a particular service.

The first step is to declare each distinct public service of a component as an instance of Service class. Fig. 2 shows how to declare the service to transfer money (*ExpressBankTransferService*) as a DAML-S service.

service:Service	
rdf:ID	ExpressBankTransferService
service:presents	rdf:resource=&bank_profile;#ProfileBankTransferringService
service:describedBy	rdf:resource=&bank_process;#BankTransferProcessModel
service:supports	rdf:resource=&bank_grounding;#BankTransferGrounding

Fig. 2. Declaration of Transfer as a DAML-S Service.

The Service class is at the top of the DAML-S ontology. Service properties at this level are very general. The property *presents* refers to the profile that describes the service capabilities; the property *describedBy* refers to the service model that describes how the service works; and the property *supports* refers to the description of how to invoke the service, given by the grounding.

A service profile provides a high level description of a service and its provider (the service profile for the transfer service in the example of Fig. 3). It is used to request or advertise services for service discovery and registry purposes. Service profile consists on three types of information: a description of the service and the service provider (in this case, the Web); the functional behavior of the service; and several functional attributes for automated service selection. In the service profile input and output parameters are described, and additional information, such as the response time, or the cost of the service.

Notice that service providers use the service profile for advertising purposes, but software agents also use this profile to specify their needs and expectations. For instance, a provider may advertise a software component that provides services for accessing an auction house catalog, whereas a software agent may also look for an auction service but with a cheaper access. Since in our approach we consider that agent functionality may be provided by an external Web service or COTS component, before registering a software component we launch a component discovery task. The software agent fulfills a service profile for the searching, demanding a specific functionality including also some requirements about quality of service (QoS). DAML-S service

profiles focus on the representation of what the service does rather than where to find the service, which improves service discovery.

The profile of the bank transfer file (see Fig. 3) contains three input parameters (the amount to be transferred, the target account number, and the user personal identification), and one output parameter (the invoice).

profileHierarchy:BankOnLine	
rdf:ID	ProfileBankTransferringService
service:presentedBy	rdf:resource=&bankService;#ExpressBankTransferService
profile:contactInformation	
profile:qualityRating	
profile:input (3)	
	{ } profile:ParameterDescription
1	profile:ParameterDescription rdf:ID=Amount
2	profile:ParameterDescription rdf:ID=TargetAccountNumber
3	profile:ParameterDescription rdf:ID=SignInInfo
profile:output	
	profile:ParameterDescription rdf:ID=ENotification

Fig. 3. Bank transfer service profile in DAML-S.

The third step is to provide a declarative description of the service properties, that is, the process model. Actually, the process model describes the individual programs that comprise the service, and conceives each program as an atomic process, a simple or a composite process. In DAML-S, a non-descomposable program is described as an atomic process, which is characterized by its ability to be executed by a single call, as methods in a software component interface.

process:ProcessModel			
rdf:ID		ExpressBankTransferProcessModel	
service:describes		rdf:resource=&bank;#ExpressBankTransferService	
process:hasProcess		rdf:resource=#ExpressBankTransfer	
daml:Class			
rdf:ID		ExpressBankTransfer	
rdfs:subClassOf		rdf:resource=#process;#AtomicProcess	
rdf:Property (2)			
rdf:ID		{ } rdfs:subPropertyOf	{ } rdfs:domain
1	TransferAmount	rdfs:subPropertyOf rdf:resource=&process;#input	rdfs:domain rdf:re
2	TransferSignInInfo	rdfs:subPropertyOf rdf:resource=&process;#input	rdfs:domain rdf:re

Fig. 4. Transfer service process model in DAML-S

Accordingly, the transfer service is described in the process model as an atomic process, as Fig. 4 shows: Each process has a set of subproperties to describe input and output parameters (for example the properties *TransferAmount* and *TransferSignInInfo* in Fig. 4). DAML-S let us to describe conditional outputs, to express, for example, when you search for product in a catalog, and the output may be a detailed description, or may be an “unclassified product” response.

Finally, the grounding is given by the WSDL description. The grounding details about service access can be considered as a mapping from the general description given in the process model to a concrete implementation of services. Thus, the Web service developer can reuse the WSDL description of the Web service as the grounding. Specifics details for this relation are given in [18].

We want to point out that using DAML-S as interface description language different types of software components, such as COTS components, Web services, or in-house components, can provide a unified description of their interface. They only differ in the grounding specification, which depends on the component implementation.

4 Service Invocation

Suppose a software agent that is participating in an electronic auction on behalf of a user. Suppose also that the auction rules say that if the auction winner has not enough money in the account, he or she will be sacked the winner would be the second finalist. Hence, the negotiation is not finished until the money transfer is done. Generally the final payment in e-markets is done by credit card, but in this case is necessary to connect to the bank Web site to perform the money transfer. For security issues, the operations over a bank account (consult, transfer) are available only through a Web service offered by the bank Web site. This way, the server can establish a secure connection to keep the data protected.

Requests to agent functionality are done during a negotiation protocol execution that controls the participation of the agent in a given interaction. Inside the agent, a protocol connector coordinates each interaction. Every time a message is received, this connector changes its state and executes a set of actions, implemented inside software components registered in the agent.

The collaboration diagram given in Fig. 5 shows the composition of the connector that coordinates the auction (the instance labelled as *Auction* in Fig. 5) with the Web service that realizes the money transfer (modelled as an instance of Actor and labelled as *BankWebService*). When the agent receives from the auctioneer a message informing that it has win the auction and has to transfer the money to a given bank account, the protocol connector will invoke the action to transfer the money to a given bank account (*action (Bank ,transfer, {amount,account})*) see an example in Fig. 5). The protocol connector that coordinates the interaction does not directly invoke the action, but it forwards the request to the Mediator component (step 1 in Fig. 5). This component works, in some way, as a service broker. The Mediator component will forward the service request to the component that provides that service and will send the result back to the connector. Notice that the Mediator performs the dynamic composition of components and connectors by handling the connector's output calls, and delegating them to the appropriate software component.

As we told above registered components are addressed by the identifier of the ontology they commit, and the Mediator only has to retrieve the component reference from the *ComponentInfo* class that provide that functionality (step 2 in Fig. 5). Since there can be more than one software component registered for a given ontology. The Mediator will choose the best component according to a rule. If none of the components satisfy some QoS requirement, or rule associated to a kind of interaction, then the Mediator will initiate the searching of a better component. For example, it will choose the component that offers the lower cost service, or the Web service nearest to the current agent location. This is a very powerful feature of our model that can be used to build more adaptable agents.

Fig. 5. Service Invocation Collaboration Diagram

Coming back to our example, since the Mediator has the URL of the bank Web service, it maps the general request “action” to a concrete invocation, after consulting the grounding for the selected component. In this case, the Mediator component constructs a SOAP Message, which is sent through the AEC component to the Web service (step 4 and 5 in Fig. 5). The AEC component has the capability of sending any kind of message through any transport protocol or component platform. Finally, the Mediator (again through the AEC component –step 6 in Fig. 5) will receive the result of the operation (step 7 in Fig. 5), and will pass it to the connector that continues the protocol execution (step 8 in Fig. 5).

5 Conclusions

In this paper, we propose integrating Web services by a component-based architecture for software agents that combine component orientation and separation of concerns. Domain specific functionality of agents and the coordination issue are modelled in separated entities. Functional components (e.g. Web services) are bound at runtime and composed with the corresponding negotiation protocol connector.

Due to the flexibility provided by the component orientation we are able to plug Web services into the agent functionality coordinating them to accomplish complex tasks. For composing coordination and agent behavior at runtime the agent has to match up connector required services with the offered services of registered components. In order to do that, both components and connectors must provide some knowledge about the services they provide and demand. Web services, COTS components, and in general software components provide their functionality through well-defined interfaces. We need to use a common access method independent of the implementation language of the component, and describe components interfaces using a shared vocabulary. As a

result we propose to use DAML-S for describing the provided interfaces of software component. DAML-S provides an ontology for describing the functionality of agent inner components. Now we are developing a FIPA compliant working prototype of a generic software agent using the compositional architecture showed in previous sections.

References

1. W3C Consortium, XML Schema. <http://www.w3c.org/XML/Schema>
2. W3C Consortium, Simple Object Access Protocol (SOAP). <http://www.w3.org/TR/2002/CR-soap12-part1-20021219/>
3. Microsoft, UDDI, Universal Description, Discovery and Integration of Web Services. The UDDI Whitepaper. <http://www.uddi.org>.
4. W3C Consortium, Web Service Definition Language (WSDL). <http://www.w3c.org/TR/wsdl>
5. W3C Consortium, Resource Description Framework. <http://www.w3c.org/RDF>
6. The DARPA Agent Markup Language Homepage. <http://www.daml.org>
7. M. N. Huhns, "Agents as Web Services", IEEE Internet Computing, July/August 2002
8. DAML Services. The DARPA Agent Markup Language Homepage. <http://www.daml.org/services/>. 2002
9. FIPA Services. <http://www.fipa.org/docs/input/f-in-00050/f-in-00050.html>
10. AgentCities Web Services Working Group. <http://www.agentcities.org/Activities/WG/WebServices>
11. M. Amor, M. Pinto, L. Fuentes, J.M. TROYA, "Combining Software Components and Mobile Agents", ESAW 2000, LNAI 1972, Springer-Verlag, 2000.
12. M. Amor, L. Fuentes, J.M. TROYA, "Training Compositional Agents on Negotiation Protocols". Iberagents 2002, Proceedings of the Fourth Iberoamerican Workshop on Multi-agent Systems, 2002.
13. G. Kiczales et al. "Aspect-Oriented Programming", Proceedings of ECOOP'97, LNCS 1241, Springer-Verlag, 1998.
14. O. Nierstrasz, L. Dami, "Component-Oriented Software Technology", Object-Oriented Software Composition, eds. O. Nierstrasz and D. Tsichritzis, Pp. 3–28, Prentice Hall, 1995.
15. L. Fuentes, J.M. TROYA, "A Java Framework for Web-based Multimedia and Collaborative Applications", IEEE Internet Computing, 3(2): pp. 55–64, April 1999.
16. L. Fuentes, J.M. TROYA, "Coordinating Distributed Components on the Web: An Integrated Development Environment", Software Practice & Experience, 31(3), pp. 209–233, 2001.
17. J. Hendler, "Agents and the Semantic Web", IEEE Intelligent Systems, March/April, 2001.
18. A. Ankolekar et al, "DAML-S: Semantic Markup for Web Services", in *Proceedings of the International Semantic Web Working Symposium (SWWS)*, July 30-August 1, 2001.