SOA/WS Applications using Cognitive Agents working in CArtAgO Environments

Michele Piunti
DEIS, Università di Bologna
Via Venezia 52
Cesena (FC), Italy
michele.piunti@unibo.it

Alessandro Ricci
DEIS, Università di Bologna
Via Venezia 52
Cesena (FC), Italy
a.ricci@unibo.it

Andrea Santi
DEIS, Università di Bologna
Via Venezia 52
Cesena (FC), Italy
andrea.santi6@studio.unibo.it

ABSTRACT

In this paper we propose a programming model and the supporting technologies for designing and programming Service-Oriented Architectures (SOA) in the perspective of Multi-Agent Systems and agent-oriented paradigms. In particular, the approach is meant to be useful for the design and programming of complex service-oriented systems, by providing first-class design tools to implement challenging features of nextcoming service applications. Indeed, besides autonomy, loose-coupled interactions, openness, flexibility, etc., the proposed design model promotes: (*i*) the use of a strong notion of agency, thus agents that can be defined along their mental attitudes to be fully autonomous systems, with pro-active and goal oriented abilities with respect of the achievement of their tasks; (*ii*) the definition artifact-based work environments where agents are meant to work and interact, thus enabling intelligent and cognitive agents to co-use, expose and manage web service technologies in a suitable functional fashion by the mean of special artifact-based infrastructures. Besides describing the basic concepts underlying the agent and artifact programming model for web services, a developing platform called CArtAgO-WS is introduced along with a describing example explored to show the approach in practice.

1. INTRODUCTION

Agents and Multi-Agent Systems are more and more recognised in the literature as a suitable paradigm for engineering SOA/WS systems, since they provide a conceptual and engineering background that naturally fits many complexities concerning SOA/WS at an high abstraction level [14, 16, 11]. Actually this view is also promoted by the official service-oriented model described in the official W3C's document about Web Services Architectures[1], which places agent-oriented concepts (such as autonomy, loosely coupling, message-based interaction, and so on) among the foundational notions used for defining web services. Indeed, as in the agent-oriented paradigm, *encapsulation* is the basic property for achieving service independency from the context: services encapsulate their logic, whose size and scope can vary, and can possibly encompass the logic provided by other services; in so doing, one or more services can be suitably composed into a collective service. Besides encapsulation, services are supposed to exhibit a certain degree of *autonomy* in that they must have control over the logic they encapsulate [9]. To face complexity of multiple interaction, changing contexts, coordination and cooperation/composition, inter-service relationships should minimise dependencies – in particular, control dependencies – retaining only the *awareness* of the service description (loose-coupling).

In the same research line that proposed the introduction of agents and artifact models for the design of web services [22] we here promote the agent-oriented programming approach for the implementation of complex service oriented systems. In this view, we first describe CArtAgO-WS as a general-purpose platform enabling the impementation of integrated SOA applications based on heterogeneous agents operating in shared, artifact-based work environments. Besides, we propose the adoption of a strong notion of agency (cognitive agents, i.e. programmable based on specific mental attitudes as belief and goals) in order to support pivotal properties as pro-activeness towards goals and task achievement, context awareness, situatedness towards highly dynamic contexts, reactiveness to multiple events etc. Finally, we propose the adoption of artifact-based infrastructure enabling agents to directly operate upon web services, including relating mechanisms like messaging and WS-* protocols. Like existing works in the agent literature, our approach results in integrating existing agent technologies with Web Services by providing a flexible *agent-oriented programming model* (and a supporting platform) for designing and developing SOA/WS as open MAS, and possibly exploiting heterogeneous agent models/languages interacting in the same artifact-based work environment. Differently form related approaches, however, the novelty of this work is in enabling the adoption of *cognitive* agents, i.e. agents that can be conceived and programmed along mental states mimicking an "intelligent" behavior. Whereas properties like loose coupling between several resources, comformity to WS-* technologies etc. must be guaranteed in every SOA application, cognitive agents – along with artifact-based support infrastructures – are thus intruduced as a modeling tool, in order to organise the system and ease the management of complex dynamics as the ones elicited by information rich environments, multiple asynchronous interactions, heterogeneous sources of events, etc.

The remainder of the paper is organised as follows: first, Section 2 presents background issues and related works, then the underlying agents and artifact programming model for the design of web services is described in Section 3. The CArtAgO-WS platform is presented in Section 4 and, to give a concrete taste of the approach, Section 5 shows a case study, discussing the implementation of a complex

[1]http://www.w3.org/TR/ws-arch/

SOA application based on agents working in artifact-based workspaces. Finally, in Section 6 we conclude the paper, briefly discussing final remarks and future works.

2. BACKGROUND

While Web services and SOA appear a promising solution for building interoperable, heterogeneous, and dynamic distributed systems, the definition of proper programming models is actually a main issue in research [8, 1, 9]. Where actually the mainstream proposals are either *object-oriented* or *component-oriented*, besides XML based interaction protocols, SOA does not provide a unifying programming model or methodology to support the core system design and development – which, therefore, still appears a challenging issue. Mainstream approaches are straightforward for building applications where services can be simply mapped onto classes, thus promoting the development of service applications – including both the server side and the client side – essentially as (distributed) object-based systems, where the interactions among objects are essentially based on remote procedure call mechanisms (RPC). For instance, Windows Communication Foundation (previously called Indigo) [29] and JAX-WS Specification[2] provide frameworks mapping Web Services and SOA programming concepts onto the OO programming model. Other examples are Service Component Architecture (SCA) [8], promoted by independent software vendors such as IBM, SAP, IONA, Oracle, BEA, TIBCO to cite some, and the Java Business Integration (JBI)[3], promoted by the Java Community. Approaches as the abovementioned cannot be considered adequate when complex SOA systems are of concerns [8, 16], as pivotal properties like autonomy, loose coupling, asynchronous interaction, concurrency, dynamic composability cannot be easily provided by the basic OO modeling [9].

To overcome this gap we guess that last research on agent-oriented architectures and related technologies could be fruitfully exploited to define a general-purpose *programming model* to support the design and the development of Web Services and SOA systems, in particular when complex applications are of concern. So far, several frameworks have been presented in the agent area for the design of SOA. Actually they mainly focus on the *integration* of agent platforms – in particular, FIPA-based platforms, such as JADE – with Web Services technologies [12, 17, 28, 20, 11, 31]: their design objective is mainly to find a common specification to describe how to seamlessly interconnect FIPA-compliant agent systems with W3C-compliant Web Services. The proposed solution usually adopts some kind of centralized *gateway* agent, working as a mediator for agents who aim to interact with Web Services on the one side (agents as service consumers) and for Web Service requests to be served by agents on the other side (agents as service provider). In JADE, for instance, a Web-Service Integration Gateway (WSIG) supports registration and discovery of Web Services by agents, registration and discovery of JADE agents and agent services by Web Services clients, automatic and cross-translation of UDDI directory entries into DF directory entries and viceversa, invocation of Web Services by JADE agents, and invocation of JADE agents by Web Services [12].

[2]`http://jcp.org/en/jsr/detail?id=224`

[3]`http://jcp.org/aboutJava/communityprocess/final/jsr208/index.html`

Differently to approaches proposing a unique gateway in order to address the interoperability between agents and Web Services, our approach is based on a dynamic creation and control of artifact-based facilities aimed at supporting agent activities at an infrastructural-level. As explained in the next sections, artifacts are special computational entities providing the access point to Web Services, they can be created and configured on the need and are exploitable in a functional / goal oriented fashion in order to build and consume complex SOA applications. Besides enabling interoperability between agents platforms and web services, we here mean to investigate a complementary approach by promoting the use of a strong notion of agency as well as distributed artifact-based facilities instrumenting agents work environment. In this view, our work is related also to existing approaches in literature investigating the use of goal-oriented/BDI agent technologies in the context of Web Services (see among others [6, 7, 5, 10, 30]). Nevertheless, more than in addressing specific issues like service orchestration, discovery and coordination, we here aim at providing a programming model general enough to enable agents to intelligently exploit web services. In particular we here focus on the intelligent interaction between agents and artifact-based infrastructures. This is also the main perspective under which the novelty of this approach should be appreciated with respect to existing industrial frameworks implementing SCA, including those integrating established research technologies like tuple spaces (such as in the case of GigaSpace[4]). In so doing, we aim at exploiting some the foundational features of the agent paradigm [15] to seamlessly tackle with a single computational model all those aspects that are put forth by actual SOA models. In this view we here promote the capability to integrate both a task-oriented/process-oriented behaviour – such in the case of agent based workflows [2] or goal oriented business processes [27] – and a reactive (even-driven) behaviour, such in the case of Event-Driven Architectures (EDA), which are meant to be a main aspect of SOA 2.0[5].

3. AN AGENT-ORIENTED PROGRAMMING MODEL FOR WEB SERVICES

The agent-oriented programming model for implementing Web Services relies on adopt A&A (Agents and Artifacts) that was introduced in the context of agent-oriented software engineering [18] as the reference meta-model. A&A takes inspiration in particular from human cooperative environments as they are modeled in Activity Theory and other approaches in cognitive science. In this view, a service – or an application using services, that can a be service it self – is organized in terms of a set of autonomous entities, the agents, that work together inside shared computational working environments. Agents are *pro-active* entities, i.e. they are designed and programmed so as to achieve some kind of goal or do some kind of task autonomously, encapsulating the logic and control of their activities. To achieve their design objectives agents are assumed to *perceive* and *act* upon the computational environments where they work, besides directly communicating in a message passing fashion by means of some agent communication language (ACL).

[4]`http://www.gigaspaces.com`

[5]`http://www.infoworld.com/t/architecture/make-way-soa-20-420`

Besides being pro-active, agents are also *reactive*, i.e. they are meant continuously perceive events from the environment and react accordingly in a situated way.

The computational environments – possibly distributed – are modeled in terms of *workspaces*, representing virtual locations containing sets of first-class entities called *artifacts*. In A&A terms, artifacts represent external resources and tools that agents can share and use to fulfill their work, Moreover, they are *function-oriented* entities, i.e. they are designed to encapsulate functionalities that agent can exploit to achieve their (individual and collective) objectives.

The interaction model defining agent-artifact interaction is based on the notion of *use* and *observation*, mimicking human interactions with artifacts in their human (cooperative) environments. Each artifact has a *usage interface* listing a set of controls that agents can use to trigger and execute artifact operations. Operation execution – which occur asynchronously to agent behavior – can lead to the generation of *observable events* that the agent using the artifact, and other agents possibly observing it, can perceive.

Besides events, each artifact can expose an observable state, in terms of one or multiple *observable properties* whose value can be dynamically perceived by agents, without necessarily executing operations. By *focussing* an artifact, agents can perceive artifact observable state and events without directly using it – i.e. without triggering operation processes[6].

So, from a design point of view agents and artifacts are the basic blocks to organize service-oriented systems: agents are meant to encapsulate the logic and control of tasks, activities and processes – both in the case of client applications and service applications – while artifacts are useful devices exploitable by agents to work together and to interact with external systems.

Compared to object-based and component-based programming model, here we have, on the one hand, first-class computational entities to encapsulate (and hide) threads of controls (agents), providing a direct support for building concurrent programs – exploiting parallel and distributed hardware – but without dealing with threads and low-level related mechanisms. Artifacts, on the other hand, make it possible to realize coordination mechanisms and shared resources for agents without the need to face low-level issues like synchronization mechanisms.

Besides these technical aspects, the most important value is from a methodological viewpoint, having in the programming model first-class abstractions that naturally capture high-level concepts such as goals, tasks, actions that typically appear at the business level, as well as in the modeling and design of complex services. By considering advanced services in particular, conceptually and pragmatically agents *reactivity* and *proactivity* can be important ingredients to concretely program either event-driven systems, as promoted by SOA extensions to integrate Event-Driven Architectures, either *goal-driven* services – which accounts for advanced aspects such as goal-driven service composition and orchestration [30], goal-oriented business process managements [6], and so on. Artifacts are ideal to implement those business resources that are involved in service design, including also tools (from the agent viewpoint) embedding the machinery related to legacy/standard technologies, such as Web Services protocol stack management. Notice that, differently from aproaches proposing a static, gateway based, solution, in the proposed model the set of artifacts in a workspace is dynamic: agents have proper actions to dynamically create (and dispose) artifacts as instances of artifact template, analogously to objects in object-based systems.

Finally, a main point of Web Services and SOA programming model concerns *composition*, in this case related to how a single service consumer/provider application can be assembled (and extended) by (possibly heterogeneous) parts. Component-based approaches provide a straightforward support – compared to flat OO models, in particular – by introducing assemblies and proper wiring models. In the programming model proposed in this paper, composition is totally dynamic, realized both by agents that can dynamically join and quit workspaces, and by the runtime creation (by agents) of artifacts, that can be then discovered (and disposed) at runtime by other agents on the need.

[6]A detailed description of agents and artifacts programming model is outside the scope of this paper. The interested readers can find more technical details in [26, 24].

4. THE CArtAgO-WS PLATFORM

CArtAgO-WS (Common ARtifact infrastructure for AGent Open environment and Web Services) is a platform providing a concrete programming platform and technology implementing the abstract model described in the previous section. Actually the platform integrates different kind of agent technologies (see Fig. 1).

A first technology is CArtAgO [26], as the platform / infrastructure used to develop and execute the computational worlds where agents live. CArtAgO provides both a concrete computational/programming model for developing artifacts and a runtime environment for their execution. Currently CArtAgO is fully developed in Java and also the basic API to program artifacts is Java-based[7].

Then, multiple technologies (and computational models and architectures) can be exploited to program and execute agents in CArtAgO [23, 19]. Among the other, *Jason* technology [3] – which will be used in examples in Section 5 – based on the AgentSpeak agent programming language, which provides a programming model based on the *BDI* (Belief - Desire - Intention) architecture to program so called *intelligent* or *cognitive* agents. Other integrated technologies include simpA [25], a framework to develop activity-oriented agents on top of the Java platform, and Jadex [21], another platform based on Java to develop BDI, goal-directed agents.

CArtAgO-WS makes it possible to exploit CArtAgO and agent technologies such as *Jason*, simpA and Jadex to develop service-oriented applications based on Web Services. Basically CArtAgO-WS extends CArtAgO by providing a predefined workspace called web-services: this workspace is instrumented with a basic set of artifacts that enable, on the one side, the interaction of agents with existing Web Services and, on the other side, the construction and deployment of new Web Services controled by agents. This basic support is described in the Subsection 4.1. Besides this basic enabling level, CArtAgO-WS includes an open set of artifacts implementing WS-* functionalities: this support is described in Subsection 4.2.

4.1 Basic Support

[7]CArtAgO implementation is open-source and available at `http://cartago.sourceforge.net`

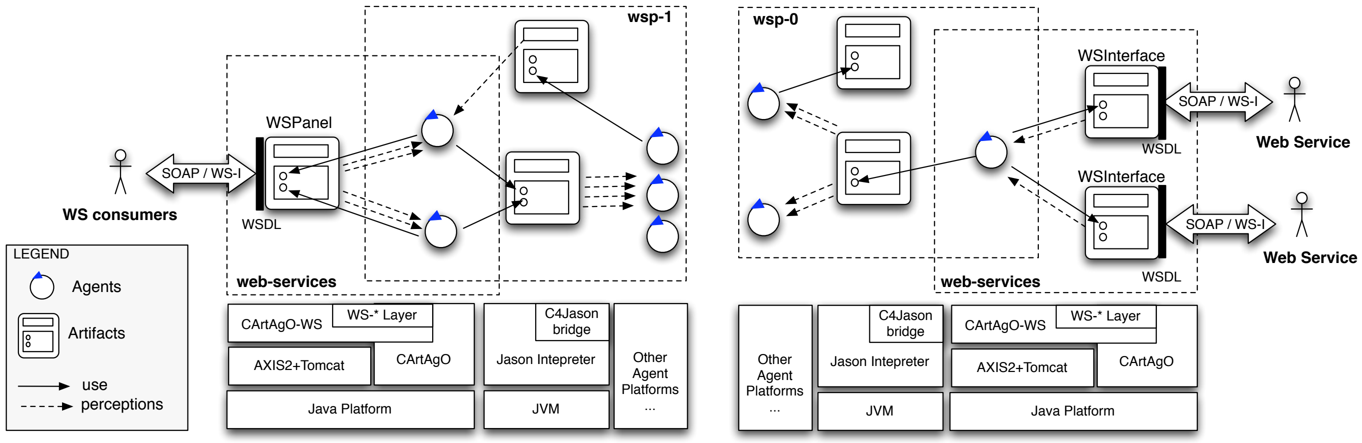

Figure 1: CArtAgO-WS platform overview. On the left side, a CArtAgO-WS node running a Web Service, composed by two workspaces—`web-services` and `wsp-1`. In `web-services` workspace, an instance of `WSPanel` artifact is shared and used by two agents to process WS requests and send replies. On the right side, a CArtAgO-WS node running an application using existing Web Services. Also in this case we two workspaces are shown—`web-services` and `wsp-0`. In `web-services` workspace two instances of `WSInterface` artifact are exploited by the same agent to interact (concurrently) with two distinct Web Services.

In CArtAgO-WS a SOA application is deployed in a web-services workspace instrumented by special artifact-based infrastructures that agents can instantiate and use to provide or consume Web Services with no disruption to the applications themselves. In particular, CArtAgO-WS provides two basic types of artifacts to enact an Agent-Web Service interaction, namely `WSInterface` and `WSPanel` artifacts (as showed in Fig. 1). They work either as wrappers for exploiting and integrating low-level WS enabling technologies, either as means to *reify* in the agents' world resources and tools dwelling Web Services.

To work with a given Web service, an agent instantiates a `WSInterface` artifact specifying its WSDL document – which describes the service to interact with – and optionally other details such as the specific name/port type to be used (if the WSDL includes multiple port types and services), and a local name representing the endpoint to which the artifact is bound to receive messages (e.g. replies). Once created, `WSInterface` provides basic functionalities to interact with the specified Web Service, in particular to send messages for executing operations and to get the replies sent back by the service, according to the message exchange patterns defined in the WSDL and to the quality of service specified by the service policies (in particular, security and reliability). To interact with multiple Web Services, multiple `WSInterface` artifacts must be created, one for each service: agents can then use such artifacts to interact with the services concurrently. Different agents can also use the same `WSInterface` artifact to interact with the same service. `WSInterface` usage interface includes general purpose operation controls aimed at enabling the interaction with any possible Web Service, according to the wide set of interaction protocols. In particular, it includes operations to send a message to the service in the context of an operation (`sendWSMsg`) and to get the reply to messages previously sent during an operation (`getWSReply`). Besides, it includes higher-level operations to directly support basic MEPs, such as the request-response (in-out) MEP (`requestOp`) which sends a request message and generates an event when the response message arrives. An additional operation is provided to configure the SOAP based interaction and for the support of basic WS-* standards (i.e. WS-Addressing).

Besides interacting with existing services, CArtAgO-WS enable agents to the creation and the management of new Web Services. In particular, the `WSPanel` artifact is provided to allow agents to create, set up and control a Web Service. Analogously to the previous case, `WSPanel` can be instantiated specifying a WSDL document related to the Web Service to be produced. Once created, `WSPanel` provides basic functionalities to manage service requests, including receiving and sending messages according to the specific MEP as described in the WSDL, and basic controls to configure security and reliability policies. Also in the case of `WSPanel`, the usage interface includes a set of general purpose operation controls enabling the interaction according to the wide spectrum of possible WS messaging patterns. Operations are available to retrieve or be notified about requests/messages arrived to the Web Service possibly specifying filters to select messages on the basis of their content/meta-data (`getWSMsg` and `subscribeWSMsgs`), and to send replies accordingly (`sendWSReply`). It is worth remarking that agents can *dynamically* create, quit and re-create both `WSPanel` and `WSInterface` once they have joined a `web-services` workspace hosted in a CArtAgO-WS node: this enables the capability of dynamically deploy and re-configure Web Services not by human intervention but by agents activities, thus promoting an automated management of web-services. Accordingly, it is possible to instantiate multiple Web Services at the same time, i.e. by creating multiple `WSPanel` artifacts, one for each service. Also in this case, multiple agents can use the same `WSPanel` artifact to process the service requests and/or to cooperatively manage the message exchange pattern related to individual requests.

4.2 WS-* Layer Support

CArtAgO-WS has been conceived to be modularly extended to support the full spectrum of Web Services stack protocol, implementing in particular those specifications that appear in the WSIT (Web Services Interoperability Technologies) set [8]. In the following, we refer to such module of CArtAgO-WS as WS-* layer.

Generally speaking, the objective of the WS-* layer is to provide a straightforward way – for developers and agents, finally – to get access, and manage all the processing related to WS specifications. Part of the functionalities are provided directly by `WSInterface` and `WSPanel` artifacts, and part are provided by additional kinds of artifacts, namely `Wallet` and `WSRequestMediator`

First, `WSInterface` and `WSPanel` artifacts provide specific operations to configure them so to support specific policies related to aspects such as security (WS-Security), reliability (WS-ReliableMessaging), coordination (WS-Coordination). In that way, interface and panel artifacts automatically enrich/process/validate SOAP messages with all the necessary information to adhere to the WS specification.

In some cases, however, the information needed to configure `WSInterface` and `WSPanel` artifacts could be quite articulated and require protocols for their construction. It is the case, for instance, of WS-Coordination (WS-C). To this end, `WSRequestMediator` (RM) artifacts are provided, to be used by agents to retrieve (or create) dynamic information required by complex specification such as WS-C. For instance, suppose that an agent aims to create a new WS-AtomicTransaction (WS-AT) interaction. To this end, the agent can use a RM to (create and) retrieve a coordination context, properly configured following WS-Coordination and WS-AT standards. Actually, to achieve these functionalities, the WS-* layer contains also proper worker agents – working behind RM artifacts, not visible to application agents – that are responsible of engaging communication protocols eventually needed to create context information and finally fulfill the agent requests.

Finally, the `Wallet` artifact is introduced as "personal artifact" of agents interacting with Web Services, so to ease the management of profile/context information eventually needed by WS specification and retrieved or created by means of RM artifacts. Such information can range from security tokens as required by WS-Security to dynamic coordination contexts used in WS-C protocols. In a typical scenario, a WS consumer agent first gets profile information from the wallet and then uses it to configure the `WSInterface` adopted to interact with the Web Service.

5. A CASE STUDY: *BOOK AN HOLIDAY* SCENARIO

After providing an abstract account of the main elements of CArtAgO-WS, in this section we show them in practice by considering a case study involving some of the motivating elements at the basis of our approach. The described scenario is inspired by a typical example used in SOA/WS contexts: a client agent wants to book a holiday by exploiting a series of web services providing the required resources as hotel reservation, transport facilities, payment and so on. As an additional element of the scenario, we imagine for the client the possibility to be further notified whether a selected range of date has become available for additional reservations. This allows clients to express an interest for a given date, and thus to re-try the booking activity whether the provider signals a last minute availability (i.e. due to some reservation cancelation performed by other clients). On these basis, the involved services need to shape their activities based on situated conditions:

- A given transaction can have success, or not, given the resources which are *actually* available.
- The same transaction can be retried, based on changed contexts for which, at the moment of the first attempt, the provider could not finalize the task.

To achieve such a flexibility, service behavior can be straightforwardly expressed in terms of goals (i.e. to book an holiday, to provide reservations, etc.) and situated plans to achieve them, involving the interaction with heterogeneous resources (such as internal resources as databases, coordination and transaction facilities, other web services). Accordingly, we will design and program the service with two basic cognitive (goal-oriented) agents – programmed in *Jason* – and a basic set of artifacts, representing the heterogeneous resources needed by agents to achieve their goals.

To ease the understanding of the approach, we first briefly introduce some main concepts of *Jason* agent programming model in Subsection 5.1; then, in Subsection 5.2, we describe the design and implementation of the booking application, on top of CArtAgO-WS.

5.1 *Jason* Agent Programming Model

An agent program in *Jason* is defined by an initial set of beliefs, representing agent's initial knowledge about the world, a set of goals, and a set of plans that the agent can dynamically instantiate and execute to achieve such goals. Agent plans are described by rules of the type `Event : Context <- Body` (the syntax is Prolog like), where `Event` represents the specific event triggering the plan – examples are the addition of a new belief (`+b`), a goal (`+!g`), the perception of an observable event generated by an artifact (`+ev [source(?Art)]`), the perception of an update of an artifact observable property, (`+p [artifact(?Art)]`). The plan context is a logic formula on the belief base – a belief formula – indicating the conditions under which the plan can be executed. The plan body includes basic actions to create subgoals to be achieved (`!g`), to update agent inner state – such as adding a new belief `+b` – and to work with artifacts (provided by the integration with CArtAgO). Actions in the latter case include `use`, to trigger the execution of an artifact operation, `makeArtifact`, to create a new artifact, `lookupArtifact`, to get an artifact unique identifier, `joinWorkspace`, to join a workspace, and `focus`, to start observing a specific artifact.

All the reasoning processes driving the execution of a plan reacting to events are managed at a system level, by the agent engine. From a programming perspective this means that an agent developer can define an agent's behavior by simply declaring the goal oriented activities to be performed, in terms of plans, and the situated conditions to be evaluated in order to execute the plan. A more detailed descriptions of *Jason* agent programming language is outside the scope of this paper: interested readers can find more details in [4].

[8] `https://wsit.dev.java.net/`

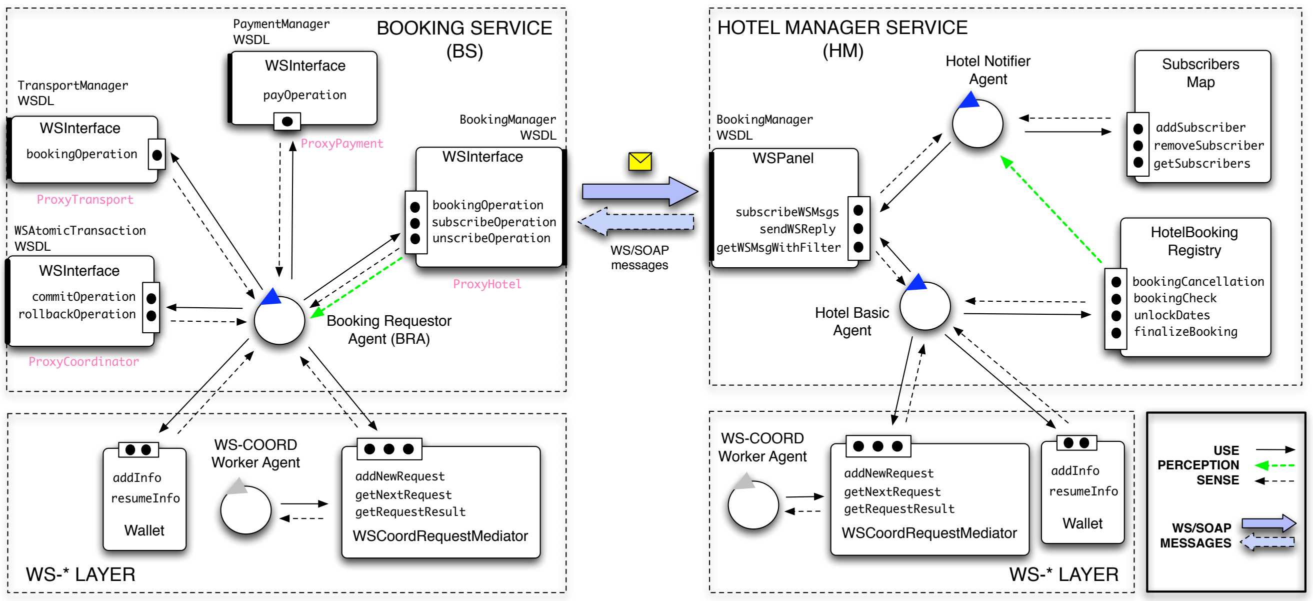

Figure 2: Structural architecture showing the services involved in the *Book an Holiday* scenario. On the left side, the *Booking Service* is controlled by a Booking Requestor Agent managing WSInterface artifacts wrapping services as *Transport Manager, Payment Manager, Hotel Manager* and *WSAtomicTransaction.* On the right side, the *Hotel Manager Service* uses two agents (*Hotel Notifier* and *Hotel Basic*) and two artifacts (*Subscribers Map* and *HotelBooking Registry*) in order to provide the booking service and the notification events exploitable by the users. The two services make use of an additional layer (on the bottom in figure) in which agents and artifacts coordinate the transactions according to WS-* mechanisms.

5.2 System Description

As showed in Fig. 2, the application is centered on two main services: *Booking Service* and *Hotel Manager*.

The *Hotel Manager Service* (HM) manages the booking tasks and also provides notification functionalities to subscribers. HM has been designed using two specialized agents, the *Hotel Basic Agent* and *Hotel Notifier Agent*, sharing and exploiting an instance of `WSPanel` to expose the service (see Fig. 2 right). The former agent manages the requests related to bookings and cancelations, exploiting – to this end – the functionalities provided by an `HotelBookingRegistry` artifact. The second agent manages the HM's notification functionalities: it uses a `SubscribersMap` artifact to keep track of the subscriptions requested and monitors the `HotelBookingRegistry` so as to notify interested subscribers as soon as changes regarding date availabilities are observed.

On the user side, the *Booking Service* (BS) realizes the task related to a client agent who wants to organize an holiday. The service is built around the pivotal role played by a *Booking Requestor Agent* (BRA), whose final goal is to plan the required reservation related to an holiday for a given period.

To achieve this goal, BRA is assumed to compose several resources, in this case related to the use of artifacts embedding external web services (see Fig. 2 left): In this case, the *Hotel Manager* service (HM) is used to *(i)* check the availability of hotel rooms for the specified period, *(ii)* subscribe for possible notifications (in case of missed availability) and *(iii)* finalize the reservation. Besides HM, the Booking Service uses additional services to accomplish its goal. In particular, a *TransportManager* service (TM) is needed to manage the booking for the transports used for arriving to (and leaving from) the specified destination. A *PaymentManager* service (PM) is used to manage bank accounts and to finalize the payment.

To execute transactions, the *Booking Requestor Agent* also exploits the support provided by the WS-* layer (a wallet and a requestor mediator artifacts). The task is managed through an atomic transaction (WS-AT) between a booking requestor and a set of WS realizing the *booking application.* The WS-AT is coordinated through a `Coordinator Service` embedded into the WS-* layer, but an external one can be used as well.

Tab. 1 shows a *Jason* cutout of BRA agent. For simplicity, we here report only the core part of the agents[9]. Agent's activities not reported here concerns further interaction between the BS and additional services. Among the others, the plan `retrieveDate` is executed to retrieve the information provided – for instance – by a human user, and to store it in form of beliefs supporting agent's behavior (goal supporting beliefs).

An example of possible dynamics follows. BRA requests to the WS-* layer the creation of a new WS-AT for managing the booking. Then BR contacts the described WS for accomplishing the booking operations. We may imagine that the hotel has already reached the number of maximum reservations – as this information has been stored in

[9]The complete source code as well as the WSDLs of the employed services are available at: `https://cartagows.svn.sourceforge.net/`.

```
+!doRequest
    : a_id(wsproxycoord, WSProxyCoord)
    & a_id(wallet,Wid) & a_id(wscoord_request_mediator, RequestMediator)
<-  // use RequestMediator to add a new request and to
    // get the result Context to be put in the wallet
    cartago.use(Wid, addInfo(Context));
    !start_booking(Res).

+!start_booking("request_succeeded")
   : a_id(wallet, WA)
<- !setupTools;
   !retrieveDate;
   !book_hotel.

+!setupTools
<- // look up for artifacts to be used and store their
   // identifiers as beliefs of the type a_id(name,id);

+!retrieveDate
<- // query the user to retrieve the dates of interest and
   // store them in the beliefbase as a fact of the type date(Dates).
   ...

+!buildWSInterface(Name, WsdlURI, Op, Port, WSInt)
   : a_id(wallet,Wid)
<- cartago.createWSInterface(Name, Wsdl, Op, Port, WSInt);
   cartago.use(Wid, resumeInfo, sensor0);
   cartago.sense(sensor0, info(Context), 1000);
   cartago.use(WSInt, configure(Context)).

+!book_hotel
   :  date(Date) & a_id(wsproxyhotel, WSProxyHotel)
<- !createBookingMessage(hotel, Date, MsgBookHotel)
   cartago.doRequestResponse(WsProxyHotel,
            bookingOperation(MsgBookHotel), 10000, ResponseHotel);
   !inspect_hotel_response(ResponseHotel, Resp);
   !book_accessories(Resp).

+!book_accessories("available")
   :   a_id(wsproxyhotel, WSPT)  & a_id(wsproxypayment, WSPP)
   & date(Date) & hprice(HotelP) & tPrice(TransportP)
<- !createBookingMessage(transport, Date, MsgTr);
   cartago.doRequestResponse(WSPT, bookingOperation(MsgTr), 10000, RespT);
   !createPayMessage("1", (HotelP+TransportP), MsgOp)
   cartago.doRequestResponse(WSPP, payOperation(MsgOp), 10000, RespP);
   !inspect_acc_responses(RespT,RespP, Result);
   !finalize(Result).

+!book_accessories("not_available")
   :  a_id(wsproxyhotel, WSPH) & dates(Dates)
<- !createSubscribeMessage(subscription, Dates, MsgSubscription);
   cartago.focus(WSPH);
   cartago.use(WSPH, subscribeOperation(MsgSubscription) ).

+dateNotMoreFull(DateId) [source(WSPH)]
   :  a_id(wsproxyhotel, WSPH)
<- cartago.stopFocusing(WSPH);
   !book_hotel;
```

Table 1: *Jason* cutout of the Booking Requestor Agent.

the `HotelBookingRegistry` artifact – for some of the dates in the requested period. In that case, the HM replies to BRA with a message notifying the inability to finalize the reservation. As BRA retrieves this information from the `WSInterface` artifact related to the HM, and it is *forced* to rollback the WS-AT. In the hope that some client will cancel a reservation for the desired date, BRA can now use the HM `WSInterface` for subscribing itself for the notification of possibly further availability. Then, by focusing the HM `WSInterface`, BR waits for a possible HM's notification. BR's subscription is now handled within the HM service by the *Hotel Notifier Agent*, which stores the request in the `SubscribersMap` artifact (see Fig. 2 right and Tab. 2 for the *Hotel Notifier* implementation). If some other agent interacting with HM cancels its reservation for the subscribed date, such a change is signalled, within the HM side, to the `HotelBookingRegistry` artifact which stores the data related to the various reservations. In this case, the *Hotel Notifier Agent* is supposed to receive a signal from the registry (`+data_status_changed(Date, DateStatus)`). As soon as it perceives the *+data_status_changed* event, it creates a new subgoal to process such information, by retrieving the subscribers matching the given date, and by sending back a notification message to the booking service who subscribed

```
+!run_hotel_service_notifer_agent
<-  !setupTools;
    !doSubscription.

+!setupTools
<-  // look up for artifacts to be used and store their
    // identifiers as beliefs like a_id(name,id).

+!doSubscription
     : a_id(wspanel, WSPanel) & filter(Filter)
<-   cartago.newObj("alice.cartagowsapi.WSMsgBasicFilter",
            ["SubscribeOperation|UnscribeOperation"], Filter);
     cartago.use(WSPanel,subscribeWSMsgsWithFilter(Filter)).

+percept(EvMsg) [source(MapId)]
     : a_id(subscribersMap, MapId)
<-   cartago.callObj(Ev, getOperationName, Type);
     !process(EvMsg, Type).

+!process(EvMsg, "SubscribeOperation")
     : a_id(subscribersMap, MapId)
<-   !findDates(EvMsg, Dates);
     cartago.use(MapId,addSubscriberForDates(Dates,EvMsg)).

+!process(EvMsg, "UnscribeOperation")
     : a_id(subscribersMap, MapId)
<-   !findDates(EvMsg, Dates);
     cartago.use(MapId, removeSubscriberForDates(Dates,EvMsg)).

+data_status_changed(Date, DateStatus) [source(RegistryId)]
     : a_id(registry,RegistryId) & a_id(subscribersMap, MapId)
<-   cartago.use(MapId, getSubscribersForDate(Date),s0);
     cartago.sense(s0,subscribers(Subscribers));
     // create NotificationMsg
     !notifySubscribers(Subscribers,NotificationMsg).

+!notifySubscribers([WSMsgInfo|T],NotificationMsg)
     : a_id(wspanel, WSPanel)
<-   cartago.use(WSPanel,sendWSReply(WSMsgInfo,NotificationMsg));
     !notifySubscribers(T,NotificationMsg).

+!notifySubscribers([],NotificationMsg).

+!findDates(EvMsg, Dates)
<-  // parses EvMsg to retrieve the period of interest
    //  and unifies it with Dates.
```

Table 2: *Jason* cutout of the Hotel Notifier Agent

(*notifySubscribers* plan).

Within the booking service, the message indicating a new availability is translated by `WSInterface` artifact related to the HM service. Furthermore, that artifact automatically signals an event in form of percept to the internal BRA agent. Also in this case, the event received by BRA is an agent's percept `+dateNotMoreFull(DateId)`). It contains a date identifier by which the agent can match the event and thus recognize it as a meaningful one with respect to its goals. In so doing, the BRA can now re-execute a new *book_hotel* plan, by which the activities needed to achieve the goal are replanned from scratch. Differently from what happened in the first attempt, the BRA now succeeds to book either the hotel and the transport service, since all the required resources are actually available. Hence, it uses the PM for the payment and it commits the transaction, finally completing its task. Notice that the all mechanisms holding BRA to its idle state, during which it simply waits for a notification, as well as the mechanisms related to its re-awaken are here simply managed at a system level, both by CArtAgO-WS and *Jason*: the developer only needs to specify under which conditions the events coming from the artifacts can be exploited to reactivate the agent practical reasoning.

6. CONCLUSION AND FUTURE WORKS

Existing works applying agents in the context of Web Services are mainly devoted either to specific service issues – such as dynamic composition of services [16] – or to devise an integration between Web Services and existing agent technologies [12]. In this paper we focused instead on the programming model issue, i.e. devising a general-purpose agent-oriented programming model for designing and programming Web Services and WS-based SOA applications.

We guess that a major strength of the approach is the level of abstraction introduced to develop services and applications using services, as they can be conceived as open workspaces where dynamic sets of possibly heterogeneous agents work together by constructing, sharing and exploiting the proper artifacts. As forthcoming SOA systems may require services to intelligently handle multifaceted requirements, we guess agent and artifact abstractions may assume a pivotal role, i.e. in forging complex workflow of goal oriented activities, in handling complex course of events in a situated way, in promoting coordination, adaptiveness, cooperation and so forth through specialized artifacts supporting the WS-* mechanisms. CArtAgO-WS platform makes it possible to put in practice these concepts, allowing the use of different kind of agent technologies for implementing agents working inside workspaces, in particular intelligent agent programming platforms such as *Jason*. In this view, a primary objective of future works will be the use of the platform to investigate the synergy between goal-oriented and artifact-based technologies for the construction of complex SOA/WS systems, with aspects concerning, for instance, goal-oriented orchestration [10, 30], goal-oriented business process management [6] and autonomic SOA/WS [13].

7. REFERENCES

[1] S. Anand, S. Padmanabhuni, and J. Ganesh. Perspectives on service oriented architectures. In *2005 IEEE International Conference on Service Computing*, volume 2. IEEE, 2005.

[2] M. Banzi, G. Caire, and D. Gotta. Wade: A software platform to develop mission critical. applications exploiting agents and workflows. In *AAMAS Industry Track*, 2008.

[3] R. Bordini and J. Hübner. BDI agent programming in AgentSpeak using Jason. In F. Toni and P. Torroni, editors, *CLIMA VI*, volume 3900 of *LNAI*, pages 143–164. Springer, Mar. 2006.

[4] R. Bordini, J. Hübner, and M. Wooldridge. *Programming Multi-Agent Systems in AgentSpeak Using Jason.* John Wiley & Sons, Ltd, 2007.

[5] L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. COOWS: Adaptive BDI agents meet service-oriented computing (extended version). In *European Workshop on Multi-Agent Systems (EUMAS 2005)*, 2005.

[6] B. Burmeister, M. Arnold, F. Copaciu, and G. Rimassa. BDI-agents for agile goal-oriented business processes. In *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), Industry and Application Track.*, 2008.

[7] G. Casella and V. Mascardi. Intelligent agents that reason about web services: a logic programming approach. In A. Polleres, S. Decker, G. Gupta, and J. de Bruijn, editors, *Proceedings of the ICLP'06 Workshop Workshop on Applications of Logic Programming n the Semantic Web and Semantic Web Services, ALPSWS2006*, pages 55–70, 2006.

[8] F. Curbera, D. F. Ferguson, M. Nally, and M. L. Stockton. Toward a programming model for service-oriented computing. In B. Benatallah, F. Casati, and P. Traverso, editors, *Third International Conference on Service-Oriented Computing (ICSOC-05)*, volume 3826 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2005.

[9] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[10] M. Georgeff. Service Orchestration: The Next Big Thing. *DM Review*, 2006.

[11] D. Greenwood and M. Calisti. Engineering web service-agent integration. In *In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, pages 1918—1925, The Hague, Netherlands, 2004. IEEE Computer Society.

[12] D. Greenwood, M. Lyell, A. Mallya, and H. Suguri. The ieee fipa approach to integrating software agents and web services. In *AAMAS '07: Proceedings of the 6th international joint conference on Autonomous agents and multiagent systems*, pages 1–7, New York, NY, USA, 2007. ACM.

[13] S. A. Gurguis and A. Zeid. Towards autonomic web services: achieving self-healing using web services. *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, 2005.

[14] M. N. Hunhs. A research agenda for agent-based Service-Oriented Architectures. In M. Klusch, M. Rovatsos, and T. Payne, editors, *CIA 2006*, volume 4149 of *LNA*, pages 8–22. Springer-Verlag Berlin Heidelberg, 2006.

[15] N. R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[16] M. N. Huhns, M. P. Singh, and M. e. a. Burstein. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(6):69–70, Nov. 2005.

[17] X. T. Nguyen and R. Kowalczyk. WS2JADE: Integrating web service with jade agents. In *Service-Oriented Computing: Agents, Semantics, and Engineering*, volume 4507 of *LNCS*, pages 147–159. Springer Berlin / Heidelberg, 2007.

[18] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17 (3), Dec. 2008.

[19] M. Piunti, A. Ricci, L. Braubach, and A. Pokahr. Goal-directed interactions in artifact-based mas: Jadex agents playing in CArtAgOenvironments. In *International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT '08)*, volume 2, pages 207–213, Sydney, NSW, 2008. IEEE/WIC/ACM.

[20] A. Poggi, M. Tomaiuolo, and P. Turci. An agent-based service oriented architecture. In *AI*IA Workshop From Object to Agents (WOA-07)*, 2007.

[21] A. Pokahr, L. Braubach, and W. Lamersdorf. Jadex: A BDI reasoning engine. In R. Bordini, M. Dastani, J. Dix, and A. E. F. Seghrouchni, editors, *Multi-Agent Programming.* Kluwer, 2005.

[22] A. Ricci and E. Denti. simpA-WS: A Simple Agent-Oriented Programming Model & Technologyfor Developing SOA & Web Services. In *Proceedings of AI*IA/TABOO Joint Workshop From objects to Agents (WOA 2007)*, 2007.

[23] A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hubner, and M. Dastani. Integrating artifact-based

environments with heterogeneous agent-programming platforms. In *Proceedings of 7th International Conference on Agents and Multi Agents Systems (AAMAS08)*, 2008.

[24] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. *Multi-Agent Programming: Languages, Tools and Applications. (Eds.) 2009, Springer. ISBN: 978-0-387-89298-6*. Springer, 2009.

[25] A. Ricci and M. Viroli. simpA: An agent-oriented approach for prototyping concurrent applications on top of java. In V. Amaral, L. Veiga, L. Marcelino, and H. C. Cunningham, editors, *Proceedings of the 5th International Conference, Principles and Practice of Programming in Java (PPPJ 2007)*, pages 185–194, Lisbon, Portugal, sep 2007.

[26] A. Ricci, M. Viroli, and A. Omicini. The A&A programming model & technology for developing agent environments in MAS. In M. Dastani, A. El Fallah Seghrouchni, A. Ricci, and M. Winikoff, editors, *Post-proceedings of the 5th International Workshop "Programming Multi-Agent Systems" (PROMAS 2007)*, volume 4908 of *LNAI*, pages 91–109. Springer, 2007.

[27] G. Rimassa, M. E. Kernland, and R. Ghizzioli. Ls/abpm - an agent-powered suite for goal-oriented autonomic bpm. In *AAMAS (Demos)*, 2008.

[28] A. A. Shafiq, H. F. Ahmad, and H. Suguri. AgentWeb Gateway - a middleware for dynamic integration of multi agent system and web services framework. In *IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprise*, 2005.

[29] C. B. Steve Resnick, Richard Crane. *Essential Windows Communication Foundation (WCF): For .NET Framework 3.5*. Addison-Wesley, 2008.

[30] M. B. van Riemsdijk and M. Wirsing. Using goals for flexible service orchestration - a first step. In *Service-Oriented Computing: Agents, Semantics, and Engineering (SOCASE'07)*, volume 4504 of *LNCS*, pages 31–48. Springer-Verlag, 2007.

[31] L. Z. Varga and A. Hajnal. Engineering web service invocations from agent systems. In *In Proceedings of the 3rd International Central and Eastern European Conference on Multi-Agent Systems*, pages pages 626–635, Prague, Czech Republic, jun 2003.