Programming SOA/WS Systems with BDI Agents and Artifact-Based Environments

Michele Piunti
DEIS, University of Bologna
Cesena, Italy
Email: michele.piunti@unibo.it

Andrea Santi
DEIS, University of Bologna
Cesena, Italy
Email: andrea.santi6@studio.unibo.it

Alessandro Ricci
DEIS, University of Bologna
Cesena, Italy
Email: a.ricci@unibo.it

***Abstract*—Agents and Multi-Agent Systems are recognized in the literature as a suitable paradigm for engineering SOA and Web Service systems: however few works explore how to exploit agent programming languages – in particular those based on a strong notion of agency, such as BDI ones – for concretely developing such a kind of systems. In this paper we discuss a general-purpose programming model and a related platform for developing SOA/WS applications exploiting BDI agent technologies. In particular, in order to enable agents to exploit and manage web service technologies in a suitable functional fashion, we investigate the use of *Jason* agents – based on AgentSpeak(L) programming language – integrated with artifact-based environments – based on CArtAgO-WS framework.**

I. Introduction

Agents and Multi-Agent Systems are more and more recognized in the literature as a suitable paradigm for engineering SOA and Web Service systems, since they provide a conceptual and engineering background that naturally fits many complexities concerning SOA/WS at a high abstraction level [13], [14], [10]. Actually this view is also promoted both by the official service-oriented model described by W3C (http://www.w3.org/TR/ws-arch/) and by the OMG initiative about the definition of an agent meta-model and profile in the SOA perspective (http://www.omg.org/cgi-bin/doc?ad/2008-09-05).

In this perspective, besides being an effective meta-model to *design* SOA, we argue that the agent-oriented programming languages and technologies can be effective tools for concretely *programming* SOA and Web Services applications, in particular for those kinds of service-oriented systems that need to integrate advanced features such as autonomy, flexibility, reactiveness, asynchronous interaction management [6], [14]. Accordingly, in this paper we develop this issue by presenting and discussing an approach which exploits and integrates existing agent technologies – *Jason* agent programming language [3] and CArtAgO-WS framework [20] – into a general-purpose platform to program and execute SOA/WS applications. In particular, the approach allows for programming and running SOA/WS applications as multi-agent systems composed by agents based on Belief Desire Intention (BDI) working together in shared environments. Besides, agent working environments are instrumented with specific kinds of tools (namely, artifacts) that agent can use to interact with existing Web Services (as consumers), to implement Web Services (as providers) and to exploit higher-level service-oriented capabilities, such as WS-Coordination. In the proposed model 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.

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 [11], [15], [25]: 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 solutions usually adopt some kind of centralized *gateway*, 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) [11]. Conversely, the approach presented in this work is based on a dynamic creation and control of customized artifact-based facilities aimed at supporting agent activities at an infrastructural level. We argue that this would improve the modularity, scalability and (dynamic) extensibility of the systems.

Besides enabling interoperability between agents platforms and web services, a main objective of this work is to investigate the use of a strong notion of agency – and in particular agent programming languages supporting it – exploiting artifact-based environments to concretely design and build service-oriented systems. In this view we promote the integration of both task-oriented/process-oriented behavior – such in the case of agent based workflows [1] or goal-oriented business processes [24] – and a reactive (even-driven) behavior, such in the case of Event-Driven Architectures (EDA), which are meant to be a main aspect of forthcoming SOA. In this perspective, our work is related to existing approaches investigating the use of goal-oriented/BDI agent technologies in the context of Web Services (see, among others [5], [4], [9], [26]). The specific focus on strong agency is also the main novelty of this paper with respect our previous work [20], [19], where a SOA/WS programming model based on A&A and the related platform have been introduced.

The remainder of the paper is organized as follows: in

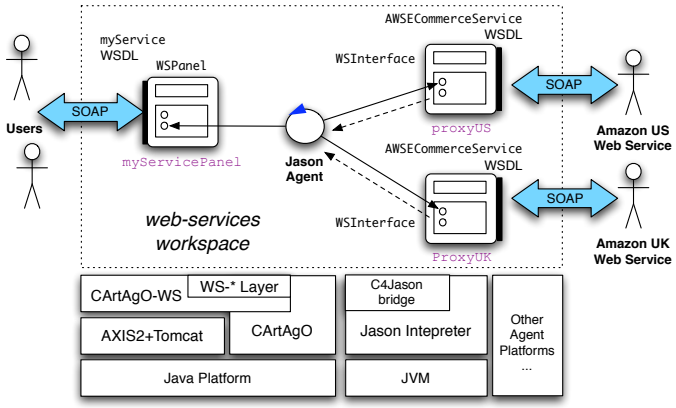

Fig. 1. CArtAgO-WS platform overview. The figure shows a CArtAgO-WS node running a Web Service built with a single agent and some artifacts deployed in the `web-services` workspace: on the right, two instances of `WSInterface` artifact are used by an agent to interact (concurrently) with two external Web Services; on the left, the agent uses a `WSPanel` artifact to provide itself a service, serving the requests coming from external WS users. In the bottom, the layers and technologies on top of which the application is built are shown.

Section II we briefly describe the basic concepts and technology on which the programming model and platform is based; in Section III a case study is discussed, where pivotal features of BDI agents as autonomy, flexibility, proactiveness and reactiveness are emphasized in a web service application involving the use of WS transactions and protocols.

II. Programming Model and CArtAgO-WS Platform

This section provides a global picture of programming model based on BDI agents and artifacts for implementing Web Services and SOA. In the next sections CArtAgO-WS platform is described, along with a simple example showing the approach in practice.

A. Agent and Artifact Programming Model for Web Services

The proposed programming model for implementing Web Services relies on A&A (Agents and Artifacts) meta-model, recently introduced in the context of agent-oriented software engineering [16]. In A&A perspective, a service – or an application using services, that can a be service itself – is organized in terms of a set of agents – as autonomous, pro-active entities – that work together inside a shared computational environment, properly designed to support their activities.

Such computational environment – possibly distributed across several nodes – is organized in terms of *workspaces* containing sets of first-class entities, called *artifacts*, representing tools and, more generally, resources that agents share and use to cooperate and fulfill their tasks. So artifacts are the basic abstraction that MAS designers and programmers can exploit to conceive and program agent environments, encapsulating functionalities that – at runtime – agents can exploit to *externalize* tasks and thus achieve their (individual and collective) objectives.

A detailed description of agents and artifacts programming model is outside the scope of this work (the interested readers can find more details in previous papers [21], [22]). Here we exploit agents and artifacts as means to design and program a SOA/WS application as a multi-agent system, in particular as workspaces where goal-oriented agents work together sharing and exploiting environment-based facilities:

- *Agents* are meant to encapsulate the logic and control of tasks, activities and business processes – both in the case of client applications and service applications;
- *Artifacts* are used to represent specialized resources and tools inside the workspaces that agents can exploit, useful in particular – in this case – to encapsulate and hide low-level aspects related to WS management.

In what follows, an integrated programming model based on BDI agents and artifacts is discussed, firstly with respect of environment setting and then with regard of agent development. In particular, *Jason* is adopted as reference programming platform for BDI agents[1]. Besides, CArtAgO-WS[2] (Common ARtifact infrastructure for AGent Open environment and Web Services) is adopted as environment programming platform.

B. Environment Side

CArtAgO-WS has been recently introduced as the reference technology for implementing SOA working environments based on the A&A model [20]. The platform integrates different modules supporting, from the one side, agent based frameworks and, on the other side, a seamless integration with Web Service technologies. As depicted in Fig. 1 (*bottom*), CArtAgO-WS is currently implemented on top of existing open-source WS technologies as Axis2 (see http://ws.apache.org/axis2/), in order to conform to the Basic Profile specification of Web Service Interoperability Organization (WS-I). The core technology of CArtAgO-WS is CArtAgO [22], which provides both a concrete computational/programming model for developing and running artifact-based environments, and API to integrate existing agent technologies (and languages, architectures) with it. This enables the implementation of MAS populated by agents possibly developed using different agent languages working together inside the same artifact-based environment.

In CArtAgO, artifacts are characterised by a *usage interface* listing a set of controls that agents can use to trigger and execute artifact *operations*, structuring artifact functionalities. Operation execution can lead to the generation of *observable events* that the agent using the artifact and other agents possibly observing it can perceive.

Basically CArtAgO-WS extends CArtAgO by providing a predefined workspace called web-services: this workspace is dynamically instrumented with different kind of specialized artifacts aimed at working with Web Services:

- *Basic artifacts*, aimed at enabling basic interactions between agents and Web Services;

[1] *Jason* is an open source platform for programming BDI agents based on AgentSpeak(L). For brevity, we omit the description of the syntax and semantic of the language: the interested reader can find more in [2] and http://jason.sourceforge.net.

[2] CArtAgO-WS is an open source platform available at: http://cartago.sourceforge.net.

- *WS-* artifacts*, aimed at supporting an enriched set of interactions, as the ones envisaged by the Web Services stack protocol;
- *Business artifacts*, aimed at providing functions for supporting agents in their business activities, as storing information which is relevant for the ongoing task in a database, wrap an external resource, control a user interface etc.

In what follows a description of the artifacts holding on the first two groups is provided, while an example of artifacts holding to the third group is given in the application of Section III.

Artifacts of the basic group allow, on the one side, agents to work with existing Web Services and, on the other side, allow the construction and the deployment of new Web Services controlled by agents. In particular, two configurable artifacts introduced are `WSInterface` and `WSPanel` artifacts (Fig. 1 shows an example of their use). To interact with an existing Web service, an agent instantiates a `WSInterface` artifact specifying its WSDL document which describes the service to interact with. Optionally it takes in other parameters such as the specific service 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, such as sending a message to the service in the context of an operation (`sendWSMsg` usage interface control) or getting 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.

Current implementation makes use of SOAP 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). In future implementation of this artifact we plan to support also resource-oriented interaction with services, as promoted by the REST architectural style [8].

To use 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.

For creating, configuring and controlling a new Web Service, a `WSPanel` artifact is provided. Analogously to the previous case, `WSPanel` can be instantiated specifying a WSDL document. Once created, `WSPanel` provides basic functionalities to manage SOAP 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 operations 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`, `getWSMsgs` 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 allows to dynamically deploy and re-configure Web Services not by human intervention but by agents activities, thus promoting an automated management of services. Accordingly, it is possible to instantiate or interact with multiple Web Services at the same time, i.e. by creating multiple `WSPanel/WSInterface` artifacts, one for each service.

Besides the basic interactions promoted by the above mentioned artifacts CArtAgO-WS introduces an additional group of artifacts. This group is included in the WS-* layer of the platform (see Fig. 1) and is aimed at supporting an extensible set of WS specifications, in particular those appearing in the WSIT (Web Services Interoperability Technologies) set (see http://wsit.dev.java.net). For doing this, the WS-* layer is instrumented with two kinds of specialized artifacts: the `WSRequestMediator` and the `Wallet`.

The `WSRequestMediator` (RM) artifact is meant to be used by agents to retrieve (or create) those dynamic information required by complex specification such as WS-Coordination (WS-C). RM's provides a general purpose usage interface so that multiple RM can be instantiated to conform to multiple protocols. For instance, suppose that an agent aims at creating a new WS-AtomicTransaction (WS-AT): to this end, an agent can use a RM to (create and) retrieve a specific coordination context, which has been previously configured following WS-Coordination and WS-AT standards.

Besides RM, a `Wallet` artifact is introduced as "personal artifact" that agents interacting with Web Services can exploit to support the management of profile/context information eventually needed by WS specification. The *Wallet* works in synergy with RM artifacts and its function is to dynamically store a portfolio of various policies which are required to conform messages to WS-* protocols. This information can range from security tokens (as required by WS-Security) to dynamic coordination contexts (as used in WS-C). In so doing a user agent can completely externalize on the wallet the management of the required policies. In a typical scenario, a agent using a Web Service first gets profile information from the *Wallet* and then uses it to configure the `WSInterface`.

So in the overall the WS-* layer allows MAS programmers to build articulated WS applications abstracting as much as possible from low-level details that concern WS-* specific protocols management (e.g. the management of the coordination contexts in WS-C), and to focus on the high-level functionalities (e.g. transactions) that agents may need to setup/exploit.

C. Agent Side

As mentioned in Subsection II-A, in our approach agents are the computational entities to specify and program the business logic of services (or applications using services). In particular, in this paper we argue that the use of a BDI agent based specification improves the abstraction by which a programmer can specify a complex task (i.e. a business process) in terms of structured agents' behavior. As agents programmable according to a BDI style can be specified by goal-oriented languages – in this paper we will use *Jason*, in particular, but analogous considerations hold by considering other agent programming languages/platforms such as 2APL [7], Jadex [18], etc. – the programming style for specifying business processes conceived as a complex chain of interleaved tasks can be natively conceived in a goal-oriented format, and thus expressed in terms of agent's plans. Then, as agents execution model is typically based on practical reasoning, BDI agents are highly adaptive in suitably finding a proper course of actions to achieve a given goal in the situated context conditions. In so doing, a main concern for programmers is to simply specify a set of behaviors realizing agent's tasks in terms of goals and plans.

Besides practical reasoning, an additional remarkable aspects is the interaction model defining interaction between agents and artifacts. From an agent point of view, artifact computational model allows two kind of interaction, as they are based on the notions of *use* and *perception*. Given this, as agents exploit artifacts to provide or use services, their programs can be expressed in native terms, i.e. by the mean of primitives for actions and perceptions. An additional remarkable aspect is related to the perceptive abilities carried out by agents with respect to artifact observable events. Indeed, due to the computational model provided by CArtAgO-WS, agents can be highly sensitive towards a rich series of events occurring upon a focused artifacts. Due to the fact that BDI model of agency typically provides constructs to explicitly handle noticeable events and react accordingly, a clear definition of events in terms of agent percepts allows a situated reactiveness of agents, mainly addressed towards the perception of relevant changes affecting the work environment, (where for relevant we refer to those information which is assumed to support the ongoing plans, as goal supporting beliefs).

To provide a concrete taste of the programming approach, the structure of a simple application is showed in Fig. 1, accounting for a service finding the best price of items – books in the specific case – by interacting with two existing web services (two Amazon Web Services, one for UK and one for US)[3]. The service is implemented by a single Jason agent, using a `WSPanel` artifact to retrieve service requests (and send responses) and two `WSInterface` artifacts to interact with the two existing Web Services.

A cutout of the agent source code is shown in TABLE I,

[3]The complete source code of the examples as well as the WSDL of the implemented services are available at CArtAgO-WS web site: http://cartago.sourceforge.net.

```
01 !find_best_price_service.
02
03 +!find_best_price_service
04  <- cartago.joinWorkspace("web-services","localhost");
05     !setupTools;
06     !cartago.use(wsPanel,subscribeWSMsgs("GetBestPrice")).
07
08 +ws_msg(Msg)
09  <- !extractMsgId(Msg,ReqId);
10     !extractItem(Msg,Item);
11     +pending_request(ReqId,Item,Msg).
12
13 +pending_request(ReqId,Item,Msg)
14  <- !prepareItemReqWs(Item, MsgReq);
15     cartago.use(awsUS,requestOp("ItemSearch", MsgReq));
16     cartago.use(awsUK,requestOp("ItemSearch", MsgReq)).
17
18 +ws_reply(ReplyMsg,ReqId)[source(From)] :
19                           pending_request(ReqId,Item,Msg)
20  <- !extractPrice(ReplyMsg,Price);
21     +price(ReqId,Item,From,Price).
22
23 +price(ReqId,Item,_,_) : price(ReqId,Item,"awsUK",_) &
24                          price(ReqId,Item,"awsUS",_)
25  <- !prepareAndSendResponse(ReqId).
26
27 +!prepareAndSendResponse(ReqId)
28  <- !computeBestPrice(ReqId,Price,From);
29     !prepareAmazonWSReply(Price,From,ReplyMsg);
30     cartago.use(wsPanel,sendWSReply(Msg, ReplyMsg));
31     -pending_request(ReqId,_,_);
32     -price(ReqId,_,"awsUK",_);
33     -price(ReqId,_,"awsUS",_).
34
35 +!setupTools
36  <- cartago.makeArtifact(wsPanel,"alice.cartagows.WSPanel",
37        ["./data/BestPriceService.wsdl"]);
38     cartago.makeArtifact(awsUS,"alice.cartagows.WSInterface",
39        [".../AWSECommerceService.wsdl"]);
40     cartago.makeArtifact(awsUK,"alice.cartagows.WSInterface",
41        [".../UK/AWSECommerceService.wsdl"]).
```

TABLE I
CUTOUT OF THE JASON AGENT SHOWED IN FIG. 1

which is briefly described in the following. The agent has a single initial goal (`find_best_price_service`, specified at line 01) and a set of plans that describe how to achieve this goal and related sub-goals. The first plan (line 03–06) is triggered as soon as the goal is instantiated, and accounts for setting up the tools needed to do the job (sub-goal `setupTools`, which triggers the execution of a plan (lines 35–41) creating the service panel, referenced by the atom `wsPanel`, and of the two WS interface artifacts, referenced as `awsUK` and `awsUS`) and subscribing the panel (operation `subscribeWSMsgs`) to receive all the message requests arriving to the service concerning the `GetBestPrice` operation.

As soon as a new WS request arrives to the panel, it generates an observable event `ws_msg(Msg)`. The agent reacts to the perception of that event (plan at lines 08–11) by extracting information about the message identifier and item to search (sub-goals `extractMsgId` and `extractItem` at line 09-10, not reported here for simplicity) and creating a `pending_request` belief containing information about the new request to process (line 11). The addition of new `pending_request` beliefs triggers the execution of a specific plan to process the requests (line 13–16). The plan accounts for using the `awsUK` and `awsUS` artifacts (lines 15 and 16) to request information about the item. The two services will answer asynchronously, with messages that are translated by the `WSInterface` in `+ws_reply(Resp,MsgId)` percepts. As

soon as replies arrive, the agent creates new `price` beliefs carrying information about the price of the item sold by the specific sources (plan at line 18–21).

As soon as both the price information from the UK service and US service are available for a specific request, the agent can prepare and send the response (sub-goal `prepareAndSendResponse`, line 25). This is done by the plan listed at lines 27–33, in which the best price is computed by the sub-goal `computeBestPrice` (not showed) exploiting the information stored in `price` beliefs and a reply message (related to the original message request) with the answer is sent back through the service panel (line 30). Finally, information about the specific request identifier and related prices are removed from the belief base (line 31–33).

Despite its simplicity, the example is meant to show how the approach allows for structuring quite naturally the business logic of the service in terms of agent plans, both to react to events occurring in artifacts populating the workspace (a `WSPanel` and the two `WSInterface` in the example) and to pro-actively execute sub-tasks on which the service business process is decomposed. By properly externalizing functionalities in artifacts, the approach makes it possible on the one side to keep agent behaviour relatively clean, purely focussed on the specification of the service logic, and, on the other side, to fruitfully exploit concurrency – artifacts execute operations in separated threads of controls – without dealing (for the agent programmer) with low-level synchronization issues. Finally, the approach promotes modularity and scalability. In the example a single agent is used to process and serve all requests arriving to the service: alternatively, a pool of agents can be used for this purpose, sharing the panel to get the requests.

III. Programming Complex Web Services using BDI Agents and Artifact-Based Environments

The benefits of adopting a BDI model of agency along with artifact-based environments are evident in particular when the design and development of complex service applications are of concerns. In what follows, Subsection III-A introduces an example application involving some of the motivating elements at the basis of the proposed approach, while Subsection III-B discuss an implementation based on CArtAgO-WS and Jason.

A. A *Case Study:* Book an Holiday *Scenario*

The described scenario is inspired by a typical example used in SOA/WS contexts: a client agent wants to book an holiday for a given date 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 goal-oriented agents, where goals are expressed in terms of specific task to achieve (i.e. to book an holiday, to provide reservations, etc.). To achieve their goals agents can organize their workflow in terms of situated plans, involving the interaction with heterogeneous resources (such as internal resources as databases, coordination and transaction facilities, other web services, etc.). Accordingly, we will design and program the involved services based on BDI (goal-oriented) agents programmed in Jason exploiting a CArtAgO-WS web-services workspace.

```
00  +!start_booking
01  <- !setupTools;
02     !retrieveDate;
03     !book_an_holiday.
04
05  +!setupTools : true
06  <- !locate_artifacts;
07     // Use the RM to request a new WS-AT and
08     // add the related ATContext into the Wallet
09     cartago.use(Wallet, addInfo(ATContext));
10     !makeInterface(proxyHM,
11           "http://webservices.hotel.com/.../BookingManager.wsdl");
12     ?artifact_id(proxyHM, ProxyID);
13     cartago.use(ProxyID, configure(ATContext)).
14
15  /* Top Level Goal */
16  +!book_an_holiday
17     : date(Dates)
18  <- !book_hotel(Dates, Res_H);
19     !book_accessories(Dates, Res_A);
20     !finalize(Res_H, Res_A).
21
22  +!book_hotel(Dates, Res_A)
23     : artifact_id(proxyHM, ProxyID)
24  <- !createBookingMessage(hotelBooking, Dates, MsgBookHotel);
25     cartago.doRequestResponse(ProxyID,
26           bookingOperation(MsgBookHotel), HotelResponse);
27     !inspect_h_response(HotelResponse, Res_H);
28     Res_H == "available".      //     fail if not available
29
30  +!book_accessories(Dates, Res_H)
31     : artifact_id(proxyTransport, TranID) & artifact_id(proxyPayment, PayID)
32     & hPrice(HotelPrice) & tPrice(TransportPrice) & bank_account_id(BankID)
33  <- !createBookingMessage(transportBooking, Dates, MsgTransport);
34     cartago.doRequestResponse(TranID,
35           bookingOperation(MsgTransport), ResponseTransport);
36     !createPayMessage(BankID, (HotelPrice+TransportPrice), MsgPay);
37     cartago.doRequestResponse(PayID,
38           payOperation(MsgPay), ResponsePayment);
39     !inspect_acc_responses(TransportResponse, PaymentResponse, Res_A);
40     Res_A == "available".      // fail if not available
41
42  /* Fail Event Handling */
43  -!book_an_holiday
44     : artifact_id(proxyHM, ProxyID) & dates(Dates)
45  <- !createSubscribeMessage(Dates, MsgSubscription);
46     cartago.focus(ProxyID);
47     cartago.use(ProxyID, subscribeOperation(MsgSubscription));
48     !finalize("not_available", "").
49
50  /* Notification from HM */
51  +dateNotMoreFull(Dates) [source(proxyHM)]
52     : artifact_id(proxyHM, ProxyID) & dates(Dates)
53  <- cartago.stopFocusing(ProxyID);
54     !book_an_holiday;
55
56  /* Finalize */
57  +!finalize(Res_H, Res_A)
58     : Res_H == "available" & Res_H == "available"
59     & wallet_entry(wsatcontext, ATContext) & artifact_id(wsProxyCoord, CoordID)
60  <- !createCommitMessage(WS-AT-Context, MsgCommit);
61     cartago.doOneWay(CoordID, commitOperation(MsgCommit));
62
63  +!finalize(Res_H, Res_A)
64     : (Res_H =/= "available" | Res_A =/= "available")
65     & wallet_entry(wsatcontext, ATContext) & artifact_id(wsProxyCoord, CoordID)
66  <- createRollbackMessage(ATContext, MsgRollback);
67     cartago.doOneWay(CoordID, rollbackOperation(MsgRollback)).
```

TABLE II
Jason CUTOUT OF THE BOOKING REQUESTOR AGENT SHOWED IN FIG. 2

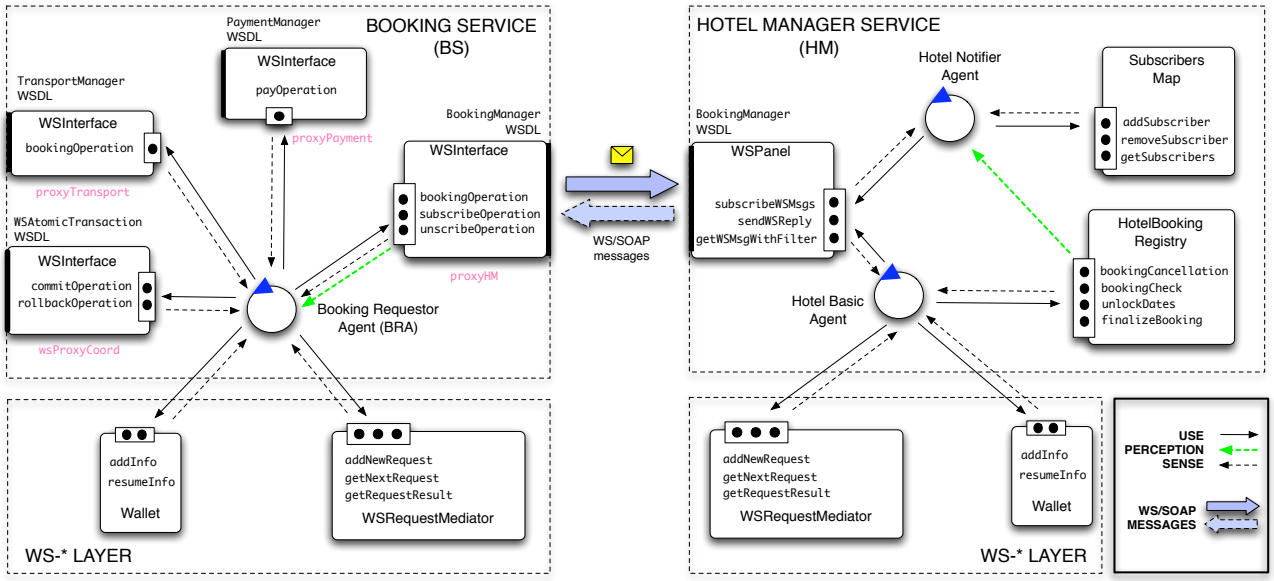

Fig. 2. Structural architecture showing the services involved in the *Book an Holiday* scenario. On the left side, the *Booking Service* is controlled by a Booking Requestor Agent managing WSInterface artifacts wrapping services as *Transport Manager, Payment Manager, Hotel Manager* and *WSAtomicTransaction*. 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 specialized agents and artifacts coordinate the transactions according to WS-* protocols.

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

To support their tasks, the agents providing the HM service use additional artifacts. In particular, in order to manage the requests related to bookings and cancelations *Hotel Basic Agent* exploits the functionalities provided by an `HotelBookingRegistry` artifact. Besides, in order to manage the HM's notification services *Hotel Notifier Agent* uses a `SubscribersMap` artifact. It is assumed to keep track of the subscriptions requested and monitor the `HotelBookingRegistry` so as to notify interested subscribers as soon as changes regarding date availabilities are observed. Notice that `SubscribersMap` and `HotelBookingRegistry` represent the external resources needed by agents to achieve their goals in the context of this specific application (i.e. business artifacts)

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 role played by a *Booking Requestor Agent* (BRA), whose final goal is to plan the required reservation related to an holiday for a given date. 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. As showed in Fig. 2, in order to externalize the computational load required to manage complex messaging, the *Booking Requestor Agent* in this case exploits the support provided by the WS-* layer (i.e., `Wallet` and `WSRequestMediator` artifacts). In addition, BRA main task is further managed through an atomic transaction (WS-AT) involving the overall set of services realizing the *booking application*. A dedicated proxy is then used to involve an external coordination service.

B. Agents and Web Services Implementation

Part of the implementation of the *Book an Holiday* Scenario is her described through the specification, in *Jason*, of the BRA agent (TABLE II shows a relevant code fragment). Agent's specification is provided in a goal-oriented format, assuming different plans addressed to a precise step in the business task to achieve.

The initial goal for BRA is to initiate a a booking activity (`+!start_booking`, line `00` in TABLE II). In so doing, BRA launch a series of sub-level goals. An initial `+!setupTools` (line `05`) is executed to retrieve or create the needed artifacts (which identifiers are stored as beliefs in the form `artifact_id(a_name, a_id)`). The WS-AT context for managing the booking is then retrieved from `RequestMediator` artifact residing in the WS-* layer, and then stored into the `Wallet` as an `ATContext` info (line `09`). A `WSInterface` artifact is also created (line `10`) for

interacting with the HM. Its artifact identifier is then stored as a belief (`artifact_id(proxyHM, ProxyID)`) and the context related to the ongoing WS-AT is used to configure it (line 13). For simplicity, a series of agent's sub-goals are here not fully specified and concerns low level computation performed for instance to manage data and to interact with additional resources. Among others, for instance, the plan `retrieveDate` (line 02) is executed to retrieve the information provided – for instance – by a human user, and to store it in form of agent's belief `date(Date)`.

As showed in Fig. 3, BRA's terminal goal is managed by a workflow of purposive activities, realized by specific plans, as they are specified by the `+!book_an_holiday` goal (line 16). The first activity consists in booking the hotel for the given dates (line 16): after having specified the context, thus retrieving the belief related to the `proxyHM`, a message for the WS request is prepared (line 24) and the HM is used by the mean of a request-response protocol (lines 25–26). We may assume that the hotel has already reached the maximum amount of reservations for (some of) the dates in the requested period (the information about date availability is stored in the HM service by the `HotelBookingRegistry`, that is an artifact implemented at the application level). In that case, the HM service replies to BS with a message notifying the inability to finalize the reservation: this message is then analyzed by a special `inspect_h_response` plan that can provide an *available* or *not_available* result. The returned literal is then matched to verify the success of the booking operation (lines 27). In so doing, a fail event will occur whether the booking operation has failed and the `Res_H` is `not_available` (line 28). Thanks to the Jason execution model, this fail event causes the root plan to fail too. Hence, the failure can be suddenly handled by a `-!book_an_holiday` plan (line 43 and Fig. 3), by which the agent can subscribe to the HM with the aim to be notified whether some new availability is signalled. So far, in the hope that some client will cancel a reservation for the desired date, the agent focuses the HM proxy (`WSInterface`) and uses it for subscribing itself for the notification of possibly further availability (lines 43–47), then waits for a possible HM's notification. In this case (line 61–65) a `+!finalize` plan is assumed to manage a rollback of the service transaction. The WS-AT is coordinated through a `Coordinator Service` which is installed in a programmable infrastructure (WS-* layer) together with the set of the services required by WS-Coordination specification.

Each BRA's subscription is handled within the HM service by the *Hotel Notifier Agent*, which stores the request in the `SubscribersMap` business artifact (the structural description of the HM service is in Fig. 2, right). If, in the meanwhile, some other agent interacting with the 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 percept from the registry: as soon as a *+data_status_changed* signal is perceived, the *Hotel Notifier Agent* creates a new sub-

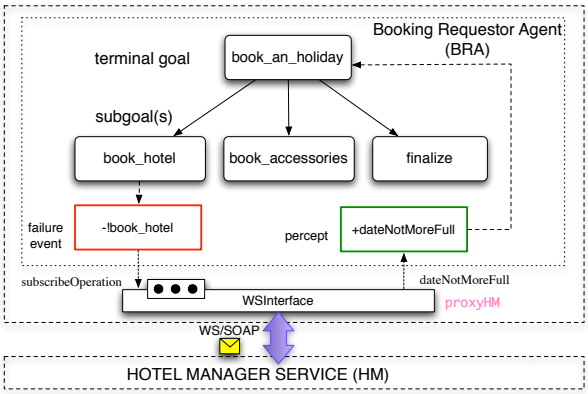

Fig. 3. Goal Decomposition Tree for Booking Requestor Agent (BRA) shows the structure of the various plans related to each sub-activity needed to achieve the terminal goal. Notice the interaction with `proxyHM` artifacts, in particular for the subscribe operation, performed after a failure in the `book_hotel` plan, and the execution of a new `book_an_holiday` plan, once a new availability is signalled by the HM.

goal to process such information, by retrieving the subscribers matching the given date, and by sending back a notification message to the BS who subscribed. Once a new availability occurs, the message coming from the HM arrives to the BS, and it is automatically translated by `WSInterface` and then it is signalled to the BRA agent. Also in this case, the event is received in form of percept and it succeeds to awaken the focusing BRA: the arriving percept `+dateNotMoreFull(Dates)[source(proxyHM)]` contains a date identifier (`Dates`) by which the agent can match the event and thus recognize it as a meaningful one, with respect to its goals (lines 36–37). In so doing, the BRA can now adopt a new instance of the *book_an_holiday* goal (line 39 and Fig. 3), by which the activities needed to achieve the goal are replanned from scratch. Differently from what happened in the first attempt, the BRA now finds the resources to succeed to book the hotel for the requested dates (HM response is, in this case, *available*).

Given this, BRA can now proceed with the following activity (`+!book_accessories`, line 30). It contacts the transport service and the payment manager, and, after having received the responses (line 33–38), it can control the results and, in so doing, achieve the terminal goal. Finally, the last activities `+!finalize(Res_H, Res_A)`, line 63) now commits the transaction upon the WS Coordination.

Some additional aspects are worth to emphasize in the described example. First, all mechanisms holding BRA to its idle state, during which it simply waits for a notification, as well as the mechanisms needed for its awaken, are here simply managed at a system level, both by CArtAgO-WS and *Jason* platforms. Once a message coming from the HM services arrives indicating an availability, the agent is suddenly and asynchronously awaken by the percept produced by the `WSInterface`. In so doing, the developer only needs to specify under which context the events coming from a given `WSPanel/WSInterface` artifact should be exploited to reactivate the agent practical reasoning. At the same time the programming model allows to automatically handle noticeable

events as failures (as in the case of missing availability in the booking task). In this case the programmer can suitably specify a purposive activity to recover the ongoing plans. The presented scenario also enlightens the support provided by the artifact based infrastructure for agent business activities. In this case the BRA only need to locate a `WSRequestMediator` and update a personal `Wallet` in order to automatically face the computational load related to the managing of the additional services needed by the protocol.

IV. Conclusion and Future Works

More and more agent technologies are recognized as a main actor in the engineering of service-oriented systems. Despite of this fact, few works have explored in literature the use of agent-oriented programming languages – and in particular those based on a strong notion of agency, such as BDI ones – to this end. In that perspective, we described a general-purpose programming model and platform for developing Web Services and SOA applications The approach promotes the adoption of BDI agents programmed with proper agent languages/platforms (Jason is used in the paper) working together in artifact-based environments (constructed with CArtAgO technology). Agents work environments are instrumented, in particular, with artifacts specialized to provide functionalities useful for exploiting (and hiding) WS protocols and related technologies (CArtAgO-WS extension).

In conclusion, a couple of aspects are worth to emphasize. First, the programming model promotes a uniform approach to design complex service/application business logic in terms of structured goal-oriented activities. Indeed, agents' practical reasoning allows, for instance, to handle complex course of events and manage failures in a situated way, promoting coordination, adaptiveness, cooperation and so forth. Second, the use of an extensible artifact-based layer makes it possible to transparently manage the computational load required for agents to conform to WS-* protocols.

Besides improving the support to WS-* technologies, a major 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 [9], [26], goal-oriented business process management [5] and autonomic SOA/WS [12].

References

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] F. Curbera, D. F. Ferguson, M. Nally, and M. L. Stockton. Toward a programming model for service-oriented computing. In *Third International Conference on Service-Oriented Computing (ICSOC-05)*, volume 3826 of *Lecture Notes in Computer Science*. Springer, 2005.

[7] M. Dastani. 2APL: a Practical Agent Programming Language. *Autonomous Agents and Multi-Agent Systems*, 16(3):214–248, 2008.

[8] R. T. Fielding and R. N. Taylor. Principled Design of the ModernWeb Architecture. *ACM Transactions on Internet Technology*, 2:115–150, 2002.

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

[10] D. Greenwood and M. Calisti. Engineering web service-agent integration. In *Proc. of IEEE Conf. on Systems, Man and Cybernetics*, 2004.

[11] D. Greenwood, M. Lyell, A. Mallya, and H. Suguri. The IEEE FIPA approach to integrating software agents and web services. In *Proc. of Autonomous agents and multiagent systems (AAMAS-07)*, 2007.

[12] 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.

[13] 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.

[14] 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.

[15] X. T. Nguyen and R. Kowalczyk. WS2JADE: Integrating web service with jade agents. In *Service-Oriented Computing: Agents, Semantics, and Engineering, vol. 4507 LNCS*. Springer, 2007.

[16] 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.

[17] M. Piunti, A. Ricci, L. Braubach, and A. Pokahr. Goal-Directed Interactions in Artifact-Based MAS: Jadex Agents Playing in CArtAgO Environments. In *Proc. of Web Intelligence and Intelligent Agent Technology (WI-IAT '08)*, Sydney, 2008. IEEE/WIC/ACM.

[18] 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.

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

[20] A. Ricci, E. Denti, and M. Piunti. A Platform for Developing SOA/WS Applications as Open and Heterogeneous Multi-Agent Systems. *Milti Agent and Gris Systems*, (to appear).

[21] A. Ricci, M. Piunti, L. D. Acay, R. Bordini, J. Hübner, and M. Dastani. Integrating Artifact-Based Environments with Heterogeneous Agent-Programming Platforms. In *Proc. of the Seventh International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 225–232, 2008.

[22] 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*, chapter Environment Programming in CArtAgO, pages 259–288. Springer, 2009.

[23] A. Ricci and M. Viroli. simpA: An agent-oriented approach for prototyping concurrent applications on top of Java. In *Proc of Principles and Practice of Programming in Java (PPPJ-07)*, 2007.

[24] G. Rimassa, M. E. Kernland, and R. Ghizzioli. Ls/abpm - an agent-powered suite for goal-oriented autonomic bpm. In *Demo Session in AAMAS 2008*, 2008.

[25] 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.

[26] 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), vol. 4504 LNCS*. Springer, 2007.