Intelligent Agents that Reason about Web Services: a Logic Programming Approach[*]

Viviana Mascardi[1] and Giovanni Casella[1,2]

[1]DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy
mascardi@disi.unige.it
[2]DMI, Università di Salerno, Via Ponte don Melillo, 84084, Fisciano (SA), Italy
casella@disi.unige.it

Abstract. The paper proposes to factor three leading edge technologies, namely Web Services, Intelligent Agents, and Computational Logic, for implementing logic-based agents that reason about interaction protocols specified using standard languages for Web Services.
A working multiagent system prototype, where agents implemented in Prolog reason about protocols expressed in WS-BPEL, has been developed.
Keywords. Intelligent Agent, Web Service, Agent-Interaction Protocol, Prolog

1 Introduction

"Agent-based systems are one of the most vibrant and important areas of research and development to have emerged in information technology in the 1990s. Put at its simplest, an agent is a computer system that is capable of flexible autonomous action in dynamic, unpredictable, typically multiagent domains. [...] Agents provide software designers and developers with a way of structuring an application around autonomous, communicative components, and lead to the construction of software tools and infrastructure to support the design metaphor. In this sense, they offer a new and often more appropriate route to the development of complex computational systems, especially in open and dynamic environments."

The above quotation, taken from [23], provides the basic definition of an agent and suggests the potential impact of adopting the agent technology for correctly engineering distributed applications working in heterogeneous, dynamic, unpredictable and open environments. A well-suited example of such an environment is the World Wide Web.

The agent technology shares many common features with another technology, that of Web Services (WSs, [21]), strongly related to the Web. WSs are software applications written in various programming languages and running on various platforms, that can both expose themselves as WSs, and use other WSs. Thus, WSs are heterogeneous, distributed, and operate in an open and dynamic environment as the Web is. Actually,

---

the recent literature in the agents' field devotes much space to exploring the relationships between agents and WSs. The most well settled opinions are that either WSs provide the infrastructure, and agents provide the coordination framework [29,10,9], or that WS and agent technologies are related by the common goal of providing tools, languages, and methods necessary for engineering systems that behave in a correct way, for example w.r.t. a given interaction protocol [6].

When we consider the second part of the quotation that opens our paper - viewing agents as a design metaphor, the relationship between agents and another well-established technology, Computational Logic (CL), suggests itself. As pointed out in [12], in the Agent-Oriented Software Engineering (AOSE) field, formal methods are used in the specification of systems, for directly programming systems, and in the verification of systems. CL can be very effective for fitting all three roles above. In fact, if an agent is specified by means of a logic-based program, a working prototype of the given specification is immediately available and can be used for early testing and debugging the specification. The distinction between specifying and directly programming an agent is thus blurred. Moreover, the model checking approach to verification can be adopted to show that the agent implementation is correct with respect to its original specification. The fervid activity in this area is demonstrated by the success of many workshops, such as CLIMA[1] and DALT[2]. Various surveys and monographic collections on this topic are also available, such as [28,15,24].

Finally, CL has been used for reasoning about interaction protocols for a long time [7,8,4,2]. Since WSs define interaction protocols, it is a very natural step to adopt CL to represent [26], compose [27,25], and select [5] them.

In this paper, we factor the three technologies of WSs, agents, and CL, for implementing logic-based intelligent agents that reason about interaction protocols specified using standard languages for WSs. In particular, our agents are implemented in Prolog, are executed within the JADE agent platform, and reason about protocols expressed in WS-BPEL. The paper is organised as follows: Section 2 describes how agent interaction protocols can be expressed in WS-BPEL. Section 3 overviews our WS-aware agents, whereas Sections 4, 5, 6, and 7 provide details for the four phases of the WS-aware agent's life, namely the translation from WS-BPEL to a Prolog representation, the generation of a Prolog program that complies with the WS-BPEL interaction protocol, the activity of reasoning about the protocol, and the actual participation to the protocol, respectively. Section 8 concludes.

2 Representing Agent Interaction Protocols in WS-BPEL

The Web Services Business Process Execution Language (WS-BPEL, [1]) is a notation, layered on top of WSDL [17], for specifying business process behaviours (business protocols) based on WSs. Using WS-BPEL it is possible to specify both executable processes, that describe the actual behaviour of a participant in a business interaction and can be executed by an engine, and business protocols, that describe the mutually visi-

---

[1] http://centria.di.fct.unl.pt/~clima/
[2] http://staff.science.uva.nl/~ulle/DALT-2006/home.html

ble message exchange of each of the parties involved in the protocol, without revealing their internal behaviour. For the purpose of this work, only business protocols are used.

Besides providing language types for defining the conversational relationship between two services (`partnerLinkType`), and activities for implementing message exchange (`invoke` and `receive`), WS-BPEL also offers activities for structuring the protocol execution flow, such as `sequence`, `switch`, `if`, and `while`.

The suitability of WS-BPEL for representing agent interaction protocols (AIPs) is demonstrated by the close relationships between the constructs offered by WS-BPEL, and those offered by one of the most popular notations for AIPs, AUML [18], summarised in Table 1. To make an example, the AUML protocol depicted in Figure 1

	AUML	WS-BPEL
Roles	`ag-name/ ag-role: ag-class` box on lifelines	`myRole` and `partnerRole` tags in Partner Links
Message	Labelled arrows between lifelines	`invoke` and `receive`
Content	Speech-act based	Unspecified
Sequence	`Weak Sequencing`	`Sequence`
Condition	`Alternative`	`Switch`
Option	`Option`	`If`
Cycle	`Loop`	`While`

Table 1. Correspondence between AUML and WS-BPEL concepts

corresponds to the WSDL and WS-BPEL documents partly shown below.

WS-BPEL specification

```
1:   <process xmlns="http://schemas.xmlsoap.org/..." ....>
2:    <partnerLinks>
3:     <partnerLink name="publisherPL" partnerLinkType="lns:SellerBuyer"
                       myRole="seller" partnerRole="buyer"/>
4:     <partnerLink name="readerPL" partnerLinkType="lns:BuyerSeller"
                       myRole="buyer" partnerRole="seller"/>
5:    </partnerLinks>
6:    <variables>
7:     <variable name="continue_1" element="lns:continue_1_type"/>
8:     <variable name="choose_2" element="lns:choose_2_type"/>
....
13:   </variables>
14:   <copy><from opaque="yes"/><to>$continue_1.value</to></copy>
15:   <while condition="$continue_1.value=true">
16:    <sequence>
17:     <receive partnerLink="publisherPL" portType="lns:publisherPT"
                     operation="RCV_Mess_1" createIstance="yes"/>
18:     <copy><from opaque="yes"/><to>$choose2.value</to></copy>
19:     <switch>
20:      <case condition="$choose2.value=1">
....
n-10:       <if condition="$condition_6.value=true"/><then>
n-9:        <invoke partnerLink="publisherPL" portType="lns:publisherPT"
                       operation="SND_Mess_15"/>
n-8:        <receive partnerLink="readerPL" portType="lns:readerPT"
                       operation="RCV_Mess_15"/>
n-7:       </then> </if>
n-6:      </sequence>
```

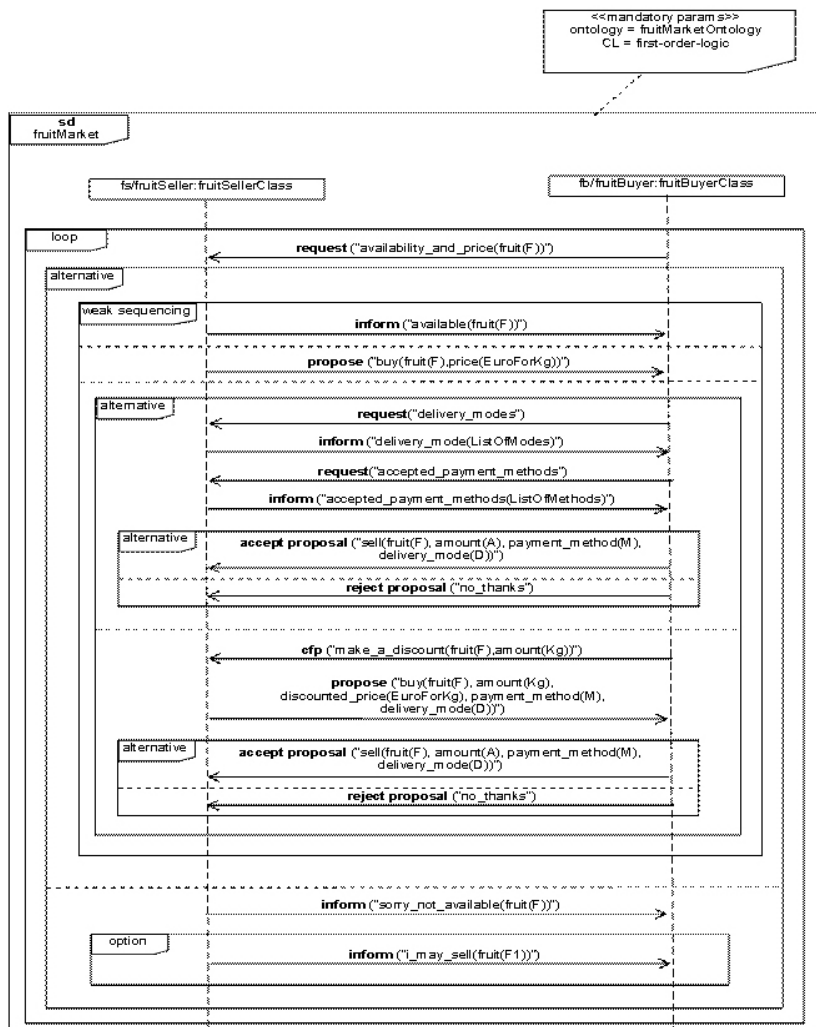**Fig. 1.** Fruit marketplace protocol in AUML

```
n-5:        </case>
n-4:       </switch>
n-3:      </sequence>
n-2:      <copy><from opaque="yes"/><to>$continue_1.value</to></copy>
n-1:     </while>
n:    </process>
```

WSDL specification

```
j:   <xs:element name="content_of_Mess_15" type="xs:string"
                      fixed="i_may_sell(fruit(F1))"/>
  ....
k:    <message name="Mess_15">
k+1:    <part name="performative" element="inform_ca"/>
k+2:    <part name="Participant" type="tns:Participant_MsgFromPublisher"/>
k+3:    <part name="Content" element="tns:content_of_Mess_15"/>
k+4:    <part name="Content_Language" element="tns:content_language_name"/>
k+5:    <part name="Ontology" element="tns:ontology_name"/>
k+6:    <part name="Protocol" element="tns:protocol_name"/>
k+7: </message>
```

For each condition to check in the AIP, a variable is defined in the WS-BPEL document (lines *6-13*), to which opaque values are associated (line *15*, where the condition of the `while` activity, corresponding to the AUML `Loop`, is given a value; line *20*, condition of the `switch` activity corresponding to the AUML `Alternative`; line *n-10*, condition of the `if` activity corresponding to the AUML `Option`). Since WS-BPEL (as AUML) does not allow to express which partner in a communication is responsible for making a condition true or false (namely, for assigning a value to a opaque variable), and since, in a very heterogeneous environment as a multiagent system is, it is not usually possible to know in advance which kind of conditions can be expressed and understood by the participants in a communication, the WS-BPEL document provides no details about conditions. The document just declares that at some point, someone will need to check a condition, and that this condition will need to be satisfied in order to allow the execution to proceed on that protocol branch (`condition = "$continue_1.value = true`, for example).

Apart from the first message of the protocol, that, according to the WS-BPEL specification [1], must be received by the service provider (the agent that publishes the document), both the point of view of the publisher and of the reader are taken into account when describing communicative actions. For example, lines *n-9* and *n-8* describe the delivery of the message identified by the number 15 from the publisher to the reader, both from the publisher's viewpoint, and from the reader's one. The WSDL document describes the details of each exchanged message: message 15 has an `inform` performative (line *k+1*) and `i_may_sell(fruit(F1))` content (line *j*).

A software tool for performing the automatic translation from AUML AIPs to WS-BPEL has been described in [11], and can be downloaded from `http://www.disi.unige.it/person/MascardiV/Software/AUML2WS-BPEL.html`.

3   Web Service-Aware Agents: a Gentle Introduction

A "WS-aware agent", whose main activities are depicted in Figure 2, is an agent able to find services, retrieve WS-BPEL documents specifying AIPs for accessing services, reason about them, and start an interaction with the document's publisher, provided that

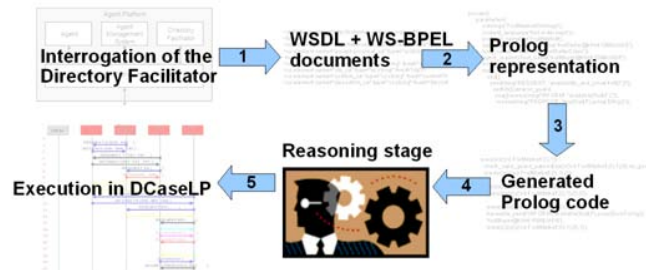the protocol satisfies the agent's desiderata. The core activities of the WS-aware agent

Fig. 2. The Web Service-Aware Agent

are implemented in tuProlog[3] [13] integrated into the JADE agent platform[4] [22] by means of the libraries offered by DCaseLP[5] [20]. The behaviour of the WS-aware agent (that sometimes we also name "reader agent", since it reads the document written and published by the service provider) is defined by the following activities:

1 The WS-aware agent exploits the facilities offered by JADE's "Directory Facilitator" in order to find the service it is interested in, and retrieves the WS-BPEL document specifying the protocol that must be followed in order to obtain the service.

2 By exploiting the JDOM technology[6], the agent translates the WS-BPEL specification of the protocol into an internal format, corresponding to a Prolog term.

3 Starting from the Prolog term, a Prolog program corresponding to a finite state machine where states are placeholders, and transitions are either communicative acts (send and receive) or conditions to check, is generated. This program generation stage is implemented in Prolog.

4 In order to verify if its desiderata (either existential or universal properties such as "There is one path where I will receive $message_1$", or "Whatever the path, I will send $message_2$") are met by the protocol, the agent exploits meta-programming facilities offered by Prolog. It makes a depth-first exploration of the SLD-tree for $P \cup \{G\}$ via R, where P is the program generated in the previous step, G is the goal that starts the execution of the protocol, and R is the leftmost selection rule. This exploration is aimed at either finding one path where the desired message is received (for demonstrating that an existential property holds), or finding a path where the final state is reached, and the expected message is not received (for demonstrating that a universal property does not hold).

5 If the desiderata of the WS-aware agent are met, the agent engages in a dialog with the publisher of the WS-BPEL document. In case the condition to satisfy was a

---

[3] http://tuprolog.alice.unibo.it/

[4] http://jade.tilab.com/

[5] http://www.disi.unige.it/person/MascardiV/Software/DCaseLP. html

[6] http://www.jdom.org/

universal property, the protocol may evolve in whatever way: the property will be satisfied by any path. Otherwise, in case of an existential property, the WS-aware agent may only try to force (as long as the decision is up to it) the execution of the path that satisfies the property, but it must take into account that at any time, the publisher might make a decision that causes another protocol branch to be followed. In this case, the WS-aware agent accepts to follow the protocol although it will not lead to the reception of the expected message. The actual execution of the protocol takes place within the JADE platform extended with the DCaseLP libraries for integrating the tuProlog interpreter.

4 From WS-BPEL to a Prolog Representation

As already anticipated, a WS-BPEL document is composed by a WSDL file and a WS-BPEL file. The WSDL file contains all the data type used to define the business process in the WS-BPEL file.

The first activity that we perform to obtain a Prolog representation of the WS-BPEL process, is to build the JDOM tree of the WSDL file. JDOM allows the programmer to represent an XML document as a tree, and to explore and modify it. In this phase, all the information about the agents and about the messages exchanged in the protocol (message sender, receiver, content, performative act) are extracted from the WSDL file. Then, a JDOM tree of the WS-BPEL file is built, and is visited following a pre-order strategy. When a WS-BPEL activity (`invoke`, `receive`, `seq`, `switch`, `loop`, etc.) is found, an appropriate Prolog representation of this activity is created. The final output is the `main_fragment(WS-BPELStructuredActivity)` term representing the protocol activities. By integrating this representation of the AIP activities with the general information about the agents and the protocol extracted from the WSDL file (protocol name, protocol parameters, etc.), the `process(Parameters, ProtocolName, PublisherData, ReaderData, MainFragment)` term is obtained. The Prolog term representing the fruit market AIP that we use as running example, is shown below.

```
process(
  %%% Ontology and message content language %%%
  parameters(ontology('FruitMarketOntology'),content_language('first order logic')),
  %%% Definition of the protocol name %%%
  protocol_name('sd FruitMarket'),
  %%% Definition of the agent publisher %%%
  agent_publisher(short_name('seller@giocas:1099/JADE'),long_name('fs/fruitSeller:
                                                      fruitSellerClass')),
  %%% Definition of the agent reader %%%
  agent_reader(short_name('buyer@giocas:1099/JADE'),long_name('fb/fruitBuyer:
                                                      fruitBuyerClass')),
  %%% Definition of the AIP activities %%%
  main_fragment(
   %%% Translation of a while activity %%%
   while(no_guard,
    %%% Translation of a sequence activity %%%
    seq([
   %%% Translation of a send activity %%%
    send(msg('REQUEST','availability_and_price(fruit(F))')),
   %%% Translation of an alternative activity %%%
    switch([
    case(no_guard,
     seq([
```

```
%%% Translation of a receive activity %%%
receive(msg('INFORM','available(fruit(F))')),
receive(msg('PROPOSE','buy(fruit(F),price(EuroForKg))')),
switch([
 case(no_guard,
 seq([
 send(msg('REQUEST','delivery_modes')),
 receive(msg('INFORM','delivery_mode(ListOfModes)')),
 send(msg('REQUEST','accepted_payment_methods')),
 receive(msg('INFORM','accepted_payment_methods(ListOfMethods)')),
 switch([
  case(no_guard,
  send(msg('ACCEPT-PROPOSAL','sell(fruit(F),amount(A),payment_method(M),
                                           delivery_mode(D))'))),
  case(no_guard,
  send(msg('REJECT-PROPOSAL','no_thanks'))])])),
 case(no_guard,
 seq([
 send(msg('CFP','make_a_discount(fruit(F),amount(Kg))')),
 receive(msg('PROPOSE','buy(fruit(F),amount(Kg),discounted_price(EuroForKg),
                                    payment_method(M),delivery_mode(D))')),
 switch([
  case(no_guard,
  send(msg('ACCEPT-PROPOSAL','sell(fruit(F),amount(A),payment_method(M),
                                           delivery_mode(D))'))),
  case(no_guard,
 send(msg('REJECT-PROPOSAL','no_thanks'))])])])),
case(no_guard,
seq([
receive(msg('INFORM','sorry_not_available(fruit(F))')),
%%% Translation of an option activity %%%
if_then(no_guard,
receive(msg('INFORM','i_may_sell(fruit(F1))')))])])])])))))
```

5   Generating the WS-BPEL-compliant Prolog Program

The philosophy behind the generation of a Prolog program starting from the Prolog
representation of the AIP, is that a finite state machine is simulated by the generated
clauses. States are meaningless terms only used to enforce the correct transitions, and
transitions correspond either to communicative actions (sending or receiving messages),
or to check of conditions. In some cases, empty transitions that just move from one state
to another, are used. Thus, the transitions can be of four kinds: *send transition*, *receive
transition*, *check transition* and *null transition*. They are exemplified below, where the
initial and final fragments of the Prolog code corresponding to the fruit marketplace
protocol depicted in Figure 1 are shown (the clause numbers written in italic are not
part of the code). Note that we did not take care of efficiency in the development of
this prototype: states can become very long terms, and no optimisations are made in
implementing the transitions.

```
clause 1:  exec(s('sd FruitMarket',0)) :-
   check_guard(s('sd FruitMarket',0), no_guard), exec(s(s('sd FruitMarket',0),0)).

clause 2:  exec(s('sd FruitMarket',0)) :- exec(s('sd FruitMarket',final)).

clause 3:  exec(s('sd FruitMarket',1)) :- exec(s('sd FruitMarket',0)).

clause 4:  exec(s(s('sd FruitMarket',0),0)) :-
  exec(s(s(s('sd FruitMarket',0),0),0)).
```

```
clause 5:   exec(s(s(s('sd FruitMarket',0),0),0)) :-
   send('REQUEST','availability_and_price(fruit(F))','seller@giocas:1099/JADE'),
   exec(s(s(s('sd FruitMarket',0),0),1)).

clause 6:   exec(s(s(s('sd FruitMarket',0),0),1)) :-
   check_guard(s(s(s(s('sd FruitMarket',0),0),1),0), no_guard),
   exec(s(s(s(s('sd FruitMarket',0),0),1),0)).

clause 7:   exec(s(s(s(s('sd FruitMarket',0),0),1),0)) :-
   exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),0)).

clause 8:   exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),0)) :-
   receive('INFORM','available(fruit(F))','seller@giocas:1099/JADE'),
   exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),1)).

clause 9:   exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),1)) :-
   receive('PROPOSE','buy(fruit(F),price(Euro))','seller@giocas:1099/JADE'),
   exec(s(s(s(s(s('sd FruitMarket',0),0),1),0),2)).

...................

clause n-3:   exec(s(s(s(s(s('sd FruitMarket',0),0),1),1),1)) :-
check_guard(s(s(s(s(s('sd FruitMarket',0),0),1),1),1),no_guard),
exec(s(s(s(s(s(s('sd FruitMarket',0),0),1),1),1),0)).

clause n-2:   exec(s(s(s(s(s('sd FruitMarket',0),0),1),1),1)) :-
exec(s('sd FruitMarket',1)).

clause n-1:   exec(s(s(s(s(s(s('sd FruitMarket',0),0),1),1),1),0)) :-
receive('INFORM','i_may_sell(fruit(F1))','seller@giocas:1099/JADE'),
exec(s('sd FruitMarket',1)).

clause n:   exec(s('sd FruitMarket',final)) :- true.
```

The predicate `check_guard(State,Guard)` (clauses *1*, *6* and *n-3*) characterises a
check transition, and succeeds if `call(Guard)` succeeds. Since no guards are specified
by the WS-BPEL AIP, their translation is always `no_guard`, and `call(no_guard)`
succeeds. The `check_guard(State,Guard)` atom may be manually edited by the
developer for inserting the application-dependent conditions that the agent might want
to check.

The predicate `send` in clause *5* (resp., `receive`, in clauses *8*, *9* and *n-1*) charac-
terises a *send transition* (resp., a `receive transition`). It is implemented by the
DCaseLP libraries, and provides an interface between tuProlog and the communication
facilities offered by JADE. It takes the message's FIPA performative [19], the content,
and the JADE address of the receiver (resp., sender), as its arguments.

The predicate that performs the generation of the code, takes the initial and final
states of the transition, the identifier of the agent, and the term representing the struc-
tured activity to translate, and returns a list of clauses that implement the transition.
The state that represents the end of the protocol is identified by the constant `final`.
The demonstration of `exec(s(ProtocolId,final))` always succeeds (clause *n*).

– *Translating cycles.* A `while(Guard,WhileActivities)` action performed in
the state `s(S,I)` for reaching the state `SFinal`, is translated into
```
clause a:   exec(s(S,I)) :- check_guard(s(S,I),Guard), exec(s(s(S,I),0)).
clause b:   exec(s(S,I)) :- exec(SFinal).
clause c:   exec(s(S,I1)) :- exec(s(S,I)).
   Clauses that translate the WhileActivities from s(s(S,I),0) to s(S,I1)
```
Our fruit market AIP starts with a while activity performed in the state `s('sd Fru-`

itMarket',0) for reaching the state s('sd FruitMarket',final). Clause *0* of our code fragment corresponds to the first clause of the translation of the while activity (clause *a*); clause *2* corresponds to clause *b*; and clause *3* to clause *c*. All the remaining activities of the protocol correspond to the `WhileActivities`, and they will need to end with reach state s('sd FruitMarket',1) (corresponding to s(S,I1) in clause *c*). When discussing the translation of options, we will see that this truly happens.

  – *Translating communication actions.* A `communication` action (where `communication` may be either `send` or `receive`) performed in the state s(S,I) for reaching the state SFinal, is translated into the clause

    clause a: `exec(s(S, I)) :- communication(Perform,Cont,Addr), exec(SFinal).`

Examples of this translation are clauses *5*, *8*, *9*, and *n-1*.

  – *Translating sequences.* A `seq([Activity0, ..., ActivityN])` action performed in the state s(S,I) for reaching the state SFinal, is translated into

    clause a: `exec(s(S,I)) :- exec(s(s(S,I),0)).`
       Clause that translates Activity0 from s(s(S, I),0) to s(s(S,I),1)
       Clause that translates Activity1 from s(s(S, I),1) to s(s(S,I),2)
       ....
       Clause that translates ActivityN from s(s(S,I),N) to SFinal

An example of this translation are clauses *4* and *5*. The state s(S, I) from which the translation starts is s(s('sd FruitMarket',0),0) (S = s('sd FruitMarket',0); I = 0) and the state to reach is s('sd FruitMarket',1). Clause *4* corresponds to clause *a*, while clause *5* corresponds to the translation of the first activity within the sequence, send('REQUEST', 'availability_and_price(fruit(F))','seller-@giocas:1099/JADE'), from s(s(S,I),0) to s(s(S,I),1). Clause *n-1* corresponds to the very last activity in the sequence. Also clauses *8* and *9* translate two items of a sequence, started in clause *7*.

  – *Translating alternatives.* A `switch([case(Guard0,Activity0), ..., case(GuardN,ActivityN)])` action performed in the state s(S,I) for reaching the state SFinal, is translated into

    clause a: `exec(s(S,I)) :- check_guard(s(s(S,I),0),Guard0), exec(s(s(S,I),0)).`
       Clauses that translates Activity0 from s(s(S,I),0) to SFinal
    ....
    clause z: `exec(s(S,I)) :- check_guard(s(s(S,I),N),GuardN), exec(s(s(S,I),N)).`
       Clauses that translates ActivityN from s(s(S,I),N) to SFinal

Clause *6* is an example of translation of a `switch` activity, and corresponds to clause *a*. Clauses *7*, *8*, *9* and successive ones correspond to the alternative branch where the fruit is available; another clause with the same head exec(s(s(s('sd FruitMarket',0), 0),1)) as clause *6*, not shown in the program fragment, corresponds to clause *z*, namely to the alternative branch where the fruit is not available.

  – *Translating options.* An if_then(Guard,Then_activities) action performed in the state s(S,I) for reaching the state SFinal, is translated into

    clause a: `exec(s(S,I)) :- check_guard(s(S,I),Guard), exec(s(s(S,I),0)).`
    clause b: `exec(s(S,I)) :- exec(SFinal).`
       Clauses that translate ThenActivities from s(s(S,I),0) to SFinal

Clauses *m-3* and *m-2* correspond to clauses *a* and *b* respectively, where s(S,I) corresponds to s(s(s(s(s('sd FruitMarket',0),0),1),1),1) and SFinal to s('sd

FruitMarket',1). *ThenActivities* correspond to the reception of the message
'i_may_sell(fruit(F1))', after which the agent moves to s('sd FruitMarket',1)
(clause *m-1*), as anticipated when we introduced the translation of cycles.

6   Is This Protocol the Right One for Me?

In order to complete the protocol execution with a success, the Prolog program of
the WS-aware agent (the fruit buyer in our running example) must contain the fact
condition_to_check(Condition), where Condition must be instantiated with ex-
ists(Action), forall(Action), or no_cond. The default for this fact is condi-
tion_to_check(no_cond), meaning that no checks on the protocol are made, but it
may be manually edited by the MAS developer if he/she wants to verify some condi-
tions. Action may correspond to one of the transition types, *send*, *receive*, *check*: the
WS-aware agent may thus check that there is one possible path where (resp. in any
possible path) a send(Performative,Content,Receiver), or a receive(Per-
formative,Content,Sender), or a check_guard(State,Guard), is executed.

Since, as already explained, for the moment we do not instantiate guards, and since
the WS-aware agent should have control over its own communicative actions, the most
interesting type of condition to check involves the reception of a message from the WS
publisher (the fruit seller of our example).

To go on with our fruit market example, let as consider a "restrictive" fruit buyer
agent that accepts to interact with the fruit seller agent only if it provides a bunch of
payment methods among which the buyer can choose. Assuming the existence of an on-
tology shared between the fruit buyer and the fruit seller, that allows them to exchange
messages whose content has been previously agreed upon, the fruit buyer agent's code
should contain the fact: condition_to_check(forall(receive('INFORM','ac-
cepted_payment_methods(ListOfMethods)',Sender))). This condition is not
verified by the protocol depicted in Figure 1 and discussed throughout the paper, as
it can be easily seen. Thus, the "restrictive" fruit buyer agent does not even start the
protocol execution.

However, a "flexible" fruit buyer might just want to check if, in the best case, the
fruit seller would allow it to choose among more than one payment method. The buyer
might force the execution of the protocol branch where this possibility takes place, as
long as the choice is up to it, but it might also be ready to accept that the seller, at some
point in the protocol execution, acts in such a way that the execution of the desired ac-
tion can no longer take place. The condition to check for the "flexible" agent would be
exists(receive('INFORM','accepted_payment_methods(ListOfMethods)',
Sender)). The protocol does verify this condition, and the agent would start the pro-
tocol execution.

Now, three situations may take place

1. The actual protocol branch executed is the one expected by the fruit buyer: w.r.t.
   our protocol, this means that the fruit seller had enough fruit of the required type to
   sell, and sends an inform(available(fruit(F))) message to the buyer. The
   buyer then requests the delivery and payment modes, and the seller provides the
   expected answer. From now on, the protocol can follow whatever branch.

2. The branch executed does not allow to verify the condition: the seller has not enough fruit to sell, and thus the branch where the buyer could perform a `receive` (`'INFORM','accepted_payment_methods(ListOfMethods)',Sender`) action can no longer be taken. The fruit buyer gives up with its hope of receiving this message, and goes on following the protocol branch determined by the seller.

3. The fruit seller sends some message that was not foreseen at all by the protocol. The buyer stops to interact with the seller, and the protocol execution fails.

While executing, the WS-aware explains its actions by printing them on a log. A fragment of this explanation in the case that everything goes as expected, is:

```
I will try to enforce the following path, as long as the choice is up to me

check_guard(s('sd FruitMarket',0),no_guard)
send('REQUEST','availability_and_price(fruit(F))','seller@giocas:1099/JADE')
check_guard(s(s(s(s('sd FruitMarket',0),0),1),0),no_guard)
receive('INFORM','available(fruit(F))','seller@giocas:1099/JADE')
receive('PROPOSE','buy(fruit(F),price(EuroForKg))','seller@giocas:1099/JADE')
check_guard(s(s(s(s(s(s('sd FruitMarket',0),0),1),0),2),0),no_guard)
send('REQUEST',delivery_modes,'seller@giocas:1099/JADE')
receive('INFORM','delivery_mode(ListOfModes)','seller@giocas:1099/JADE')
send('REQUEST',accepted_payment_methods,'seller@giocas:1099/JADE')
receive('INFORM','accepted_payment_methods(ListOfMethods)','seller@giocas:1099/JADE')

******************************* ACTUAL EXECUTION *******************************

I executed the statement check_guard(s('sd FruitMarket',0),no_guard)
I am still following the desired path!

.........

Nondeterministic action: I hoped to receive
('INFORM','delivery_mode(ListOfModes)','seller@giocas:1099/JADE')
The message that I received is the one I was waiting for!

I executed the statement
send('REQUEST',accepted_payment_methods,'seller@giocas:1099/JADE')
I am still following the desired path!

Nondeterministic action: I hoped to receive
('INFORM','accepted_payment_methods(ListOfMethods)','seller@giocas:1099/JADE')
The message that I received is the one I was waiting for!

Finally, I have reached my goal!
```

7 Let's Run!

For supporting the interaction between the Service Provider Agent and the WS-Aware Agent, we have designed and implemented the system depicted in Figure 3. In this section, we will name "Agent Services" (ASs) both the general management services offered by JADE's Directory Facilitator and by the Protocol Manager Agent that we implemented, and the application specific services such as the `fruit-selling` service offered by the fruit seller and advertised by publishing the WS-BPEL document discussed in Section 2.

The Directory Facilitator agent (DF) is provided by JADE, and offers "yellow pages" allowing agents to publish ASs, so that other agents can find and exploit them. An agent
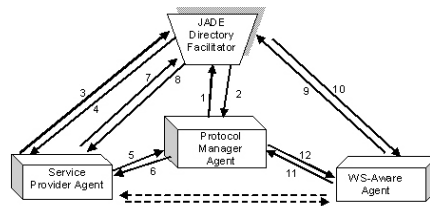
Fig. 3. Our multiagent system implemented in JADE

wishing to publish an AS must send information about itself and about the AS it provides, to the DF.

We have developed a Protocol Manager Agent (PMA) that allows agents to publish and retrieve WS-BPEL specifications representing AIPs. In other words, the PMA offers an AS that consists in the advertisement and retrieval of WSs' specifications. The PMA stores the WS-BPEL specifications received on a MySql DBMS[7].

When the JADE platform is started, the PMA registers the `protocol-publishing` and the `protocol-reading` ASs to the DF (arrows 1 and 2 in the figure 3).

When a Web Service Provider Agent (WSPA) wants to advertise an AS specified by means of WS-BPEL, it

1. looks in the DF to find the `protocol-publishing` AS (arrows 3, 4);
2. sends a message to the PMA with the WS-BPEL document, and waits to receive the protocol identifier (PID) assigned by the PMA to it (arrows 5, 6);
3. registers the AS specified by the WS-BPEL document to the JADE DF (arrows 7, 8), adding the PID obtained by the PMA and the PMA address to the AS properties.

When a WS-Aware Agents (WSAA) looks for an AS, it

1. queries the DF to find the required AS (arrows 9, 10). If a WSPA had previously registered the AS, then the WSAA obtains the name and address of the service provider, the address of the PMA, and the PID that the WSPA assigned to the AS;
2. sends a message to the PMA to obtain the WS-BPEL document representing the AIP that it must follow to obtain the AS (arrows 11, 12);
3. from the WS-BPEL specification, generates a corresponding Prolog term as discussed in Section 4;
4. generates the Prolog program from the Prolog term, as discussed in Section 5;
5. reasons about the Prolog program corresponding to the WS-BPEL AIP as discussed in Section 6; and
6. according to the reasoning outcome, eventually starts the protocol-compliant communication in order to obtain the AS (direct communication between the WSPA and the WSAA, represented by dashed arrows in Figure 3).

Figure 4 refers to an execution run of the multiagent system composed by one PMA, one WSAA, one WSPA and the DF. In this figure, the WSPA plays the role

---

[7] `http://www.mysql.com/`

67

of `fruitSeller` while the WSAA plays the role of `fruitBuyer`. The first twelve messages correspond to the communication represented by the twelve solid arrows in Figure 3, while the other messages are exchanged during the execution of the fruitMarket AIP, aimed at allowing the `fruitBuyer` to obtain the `fruit-selling` AS from the `fruitSeller`. The execution run shown here corresponds to the situation where the conditions on the protocol execution put by the buyer are all met, and the seller's proposal is accepted.
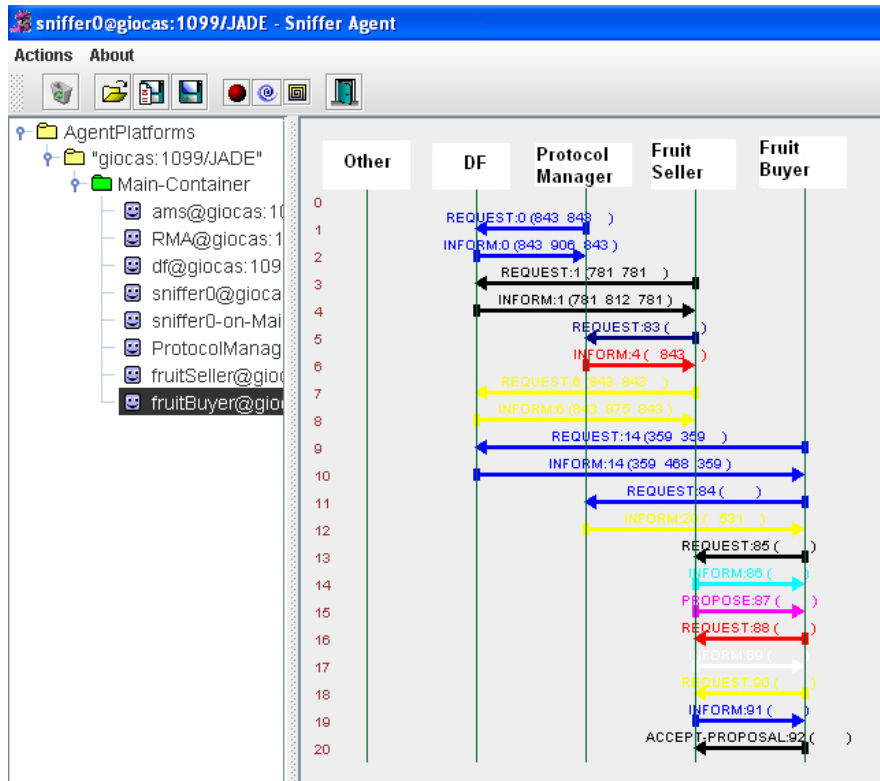

Fig. 4. Execution run in JADE

8 Conclusions

WSs are defined as heterogeneous, distributed, loosely coupled software applications. These characteristics make WSs a very flexible and scalable technology; standard languages for describing (WSDL), coordinating (WSCI, [3]), and defining business processes based on WSs (WS-BPEL) exist, and are known by most designers and develop-

ers of distributed web applications. This is one of the main reasons of the WSs' success. Despite to these advantages, WSs still remain something "static". In fact, a WS can be considered as a set of operations with predefined input and output types, unable to show any reasoning, learning, or other sophisticated capability that might allow it to dynamically adapt to the changing environment where it is situated.

On the other hand, agents are defined as autonomous, interactive, context-aware, distributed entities, working in heterogeneous, dynamic, unpredictable and open environments. Though agents are definitely more "dynamic" then WSs, their characterising features remain at a very high level, and no final agreement on standard languages for their specification and implementation has still been reached.

Many researchers today view agents either as a coordination framework for WSs or as software entities that can use WSs. In our opinion, instead, there are many applications where agents should not just use WSs, but should *extend and substitute them at all*. In fact, as pointed out by Dickinson and Wooldridge in [14], the key conceptual difference between agents and WSs is that only agents can be described in terms of human-like mental attitudes, such as Beliefs, Desires and Intentions. In those web applications where mimicking the human way of thinking may make a difference, intelligent agents built on top of the WS technology might be used instead of WSs.

This paper proposes our first step towards the usage of well established WS-related technologies to make agent-related technologies and models more concrete and effective.

The exploitation of CL makes the implementation of an "intelligent" and "human-like" behaviour easier than with other programming approaches. Although the reasoning capabilities of our WS-aware agents are currently pretty limited, the usage of CL offers to us a great potential of improvement in the near future. In fact, we are actively collaborating with other researchers involved in the Italian project MIUR PRIN 2005 "Specification and verification of agent interaction protocols" [16], aimed at proving the applicability of declarative approaches for 1) defining suitable formalisms for specifying and verifying interaction protocols, 2) developing techniques for automatic property verification and reasoning about web services, and 3) translating modelling languages into the formal languages developed in the project. We are exploring how our WS-aware agents could be implemented in dynamic linear time and/or abductive logic programming, in order to take advantage of the results already obtained by other partners in the project, in the areas of on- and off-line verification of the compliance of an agent to a protocol.

References

1. Web Services Business Process Execution Language (WS-BPEL) Version 2.0, 2005. Oasis Committee Draft 01 Sep.2005.
2. M. Alberti, D. Daolio, P. Torroni, M. Gavanelli, E. Lamma, and P. Mello. Specification and verification of agent interaction protocols in a logic-based system. In *Proc. of SAC'04*, pages 72–78. ACM, 2004.
3. A. Arkin and et al. Web service choreography interface (WSCI) 1.0. W3C Note 2002-08-08.
4. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about conversation protocols in a logic-based agent language. In *Proce. of AIIA'03*. Springer-Verlag, 2003. LNAI 2829.

5. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for customizing web service selection and composition. *Journal of Logic and Algebraic Programming, special issue on Web Services and Formal Methods*, 2006. To appear.

6. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In *Proc. of EPEW'05 and WS-FM'05*, pages 257–271. Springer-Verlag, 2005.

7. M. Bozzano and G. Delzanno. Automated protocol verification in linear logic. In *Proc. of PPDP '02*, pages 38–49. ACM Press, 2002.

8. M. Bozzano and G. Delzanno. Automatic verification of secrecy properties for linear logic specifications of cryptographic protocols. *J. Symb. Comput.*, 38(5):1375–1415, 2004.

9. L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented programming. In *Proc. of ICWI'05*, pages 205–209. IADIS Press, 2005.

10. P. A. Buhler and J. M. Vidal. Towards adaptive workflow enactment using multiagent systems. *Information Technology and Management*, 6:61–87, 2005.

11. G. Casella and V. Mascardi. AUML e WS-BPEL: Ingegnerizzare e pubblicare su web protocolli di interazione tra agenti. *Intelligenza Artificiale*, 2006. To appear.

12. P. Ciancarini and M. Wooldridge. Agent-oriented software engineering: The state of the art. In *Proc. of AOSE'00*, pages 1–28. Springer-Verlag, 2000.

13. E. Denti, A. Omicini, and A. Ricci. Multi-paradigm Java-Prolog integration in tuProlog. *Sci. Comput. Program.*, 57(2):217–250, 2005.

14. I. Dickinson and M. Wooldridge. Agents are not (just) web services: considering BDI agents and web services. In *Proc. of SOCABE'2005*, 2005.

15. J. Dix, F. Sadri, and K. Satoh, editors. Annals of Mathematics and Artificial Intelligence, 37(1-2). 2003.

16. A. Martelli et al. Modeling, verifying and reasoning about web services. Submitted to ALPSWS'06, 2006.

17. E. Christensen et al. Web Services Description Language (WSDL) 1.1, 2001. W3C Note 15 Mar. 2001.

18. M-P. Huget et al. FIPA modeling: Interaction diagrams. First proposal, 2003-07-02, 2003.

19. Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Standard 2002-12-06. `http://www.fipa.org/specs/fipa00061/`, 2002.

20. I. Gungui, M. Martelli, and V. Mascardi. DCaseLP: a prototyping environment for multilingual agent systems. Technical report, DISI, Univ. of Genova, Italy, 2005. DISI-TR-05-20.

21. H. Haas and A. Brown. Web Services Glossary – W3C Working Group Note 2004-02-11, 2004.

22. JADE Home Page. `http://jade.tilab.com/`.

23. M. Luck, P. McBurney, O. Shehory, S. Willmott, and the AL Community. *Agent Technology: Computing as Interaction – A Roadmap for Agent-Based Computing*. AgentLink III, 2005.

24. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *TPLP Journal*, 4(4):429–494, 2004.

25. S. McIlraith and T. C. Son. Adapting Golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M. A. Williams, editors, *Proc. of KR'02*, pages 482–496. Morgan Kaufmann, 2002.

26. S. A. McIlraith. Towards declarative programming for web services. In *Proc. of SAS'04*, page 21. Springer, 2004. LNCS 3148.

27. J. Rao, P. Küngas, and M. Matskin. Logic-based web services composition: From service description to process model. In *Proc. of ICWS'04*, pages 446–453. IEEE, 2004.

28. F. Sadri and F. Toni. Computational Logic and Multi-Agent Systems: a Roadmap. Technical report, Department of Computing, Imperial College, London, 1999.

29. C. Walton. Uniting agents and web services. *AgentLink News*, 18:26–28, 2005.