West2East: Exploiting WEb Service Technologies to Engineer Agent-based SofTware

Giovanni Casella¹,² and Viviana Mascardi²,*

¹DMI, Università di Salerno,
Via Ponte don Melillo, 84084, Fisciano (SA), Italy
E-mail: casella@disi.unige.it
²DISI, Università di Genova,
Via Dodecaneso 35, 16146, Genova, Italy
E-mail: mascardi@disi.unige.it
*Corresponding author

Abstract. This paper describes West2East, a Computer-Aided Agent-Oriented Software Engineering toolkit aimed at supporting the implementation of multi-agent systems. West2East exploits languages for Web Services for representing Agent Interaction Protocols (AIPs), and Computational Logic to reason about them. In particular, West2East offers libraries for the translation of AIPs represented in AUML into a) a textual notation proposed by Winikoff; b) an XML notation proposed by ourselves; c) a couple of WS-BPEL and WSDL documents; and d) a Prolog term. It also offers e) a mechanism for allowing agents that read a published AIP to reason about it before engaging in a dialog with its publisher, and f) libraries for the automatic generation of an executable program compliant to the original interaction protocol.

Keywords: Intelligent Agent, Web Service, Computational Logic, Agent Interaction Protocol, Agent-Oriented Software Engineering.

1 Introduction

The realisation of distributed, open, dynamic, and heterogeneous software systems is a challenge that involves most industries producing applications for sensors and mobile networks, pervasive and grid computing, e-commerce, e-health, and for many other domains. The Internet is often used as the communication infrastructure for exchanging both data and services among the individual applications that constitute complex systems.

Agent-Oriented Software Engineering (AOSE) studies how existing techniques can be adapted or extended in order to engineer this kind of complex distributed systems, and offers methodologies, notations, and techniques suitable for analysing, modelling, prototyping and finally implementing them following the “Multiagent System” (MAS) metaphor. On the other hand, Web Services (WSs) provide an already available and widely accepted infrastructure for supporting interoperable machine-to-machine interaction over a network, allowing heterogeneous software applications written in various programming languages and running on various platforms both to expose themselves as WSs, and to use other WSs.
Currently, many AOSE methodologies (for example, Gaia, Zambonelli et al. (2003)), notations (such as AUML, Bauer et al. (2000)) and projects (the “Flexible and Robust Protocol-Based Interaction between Agents in Open Systems” and “Societies Of ComputerS (SOCS)”, just to cite two recent ones) take interaction among agents as their starting point for engineering complex systems. To quote Ciancarini and Wooldridge (2000), in fact, “it is now widely recognised that interaction is probably the most important single characteristic of complex software.”

Unfortunately, mainly due to the lack of industrial strength Computer-Aided AOSE tools, many results achieved within the AOSE community remain confined to the academic world, while industrial practitioners continue to use classical SE approaches that often do not suit well for the requirements of nowadays’ complex applications. Web Services technologies suffer from the opposite problem: despite they are already well known and accepted by the industry, their implementation often lacks a solid engineering background that academic research might offer.

This paper describes “West2East”, a prototypical Computer-Aided AOSE tool that exploits “Web Service Technologies to Engineer Agent-based Software”, and that aims at making a step towards bridging the gap between academic research on AOSE, and industrial practice with WSs. In particular, West2East helps the systems developer in moving from the analysis and design of open MASs, carried out following AOSE methodologies, to their implementation and deployment. West2East offers facilities for enabling the implemented agents to learn how to interact with other agents and to reason about these interactions. The applicability of West2East is not limited to MASs based on the WSs technological infrastructure, although this was its initial purpose. Indeed, we have used it for implementing a MAS in the JADE agent platform, Bellifemine et al. (2000), and we envisage that its adoption in conjunction with other infrastructures and platforms may be also feasible.

West2East builds upon three technologies: Intelligent Agents, Web Services, and Computational Logic (CL).

The literature in the agents’ field devotes much space to exploring the relationships between agents and WSs. The most well settled opinions are that WSs provide the infrastructure and agents provide the coordination framework, Walton (2005); Buhler and Vidal (2005); Bozzo et al. (2005), and 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, Baldoni et al. (2005). Our position integrates both points of view. We think that, besides the coordination framework, agents also provide the metaphor that can be exploited for engineering and correctly implementing MASs based on the WSs infrastructure. Based on this claim, we propose to use AUML to model interaction protocols, and WS-BPEL – OASIS WSBPEL Technical Committees (2006) – plus WSDL – Web Services Description Working Group (2006) –, besides other notations, to share the specification of these protocols.

If we view agents as a design metaphor, the relationship between them and CL suggests itself. As pointed out in Ciancarini and Wooldridge (2000), in the AOSE field

2 http://lia.deis.unibo.it/research/socs/
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 CLIMA³ and DALT⁴. Various surveys and monographic collections on this topic are also available, such as Sadri and Toni (1999); Dix et al. (2003); Mascardi et al. (2004).

Finally, CL has been used for representing, composing, and reasoning about interaction protocols for a long time, Bozzano and Delzanno (2002, 2004); Baldoni et al. (2003); Alberti et al. (2004). In order to make complex interaction among WSs possible, it is necessary to define the sequence and conditions under which WSs exchange messages in order to perform a task to achieve a goal state. The message sequencing and associated logic, named “choreography” in technical words, may be fruitfully represented and verified adopting CL, Alberti et al. (2006). CL is also suitable for dealing with WSs composition, Rao et al. (2004); McIlraith (2004), and selection, Baldoni et al. (2006). Our usage of CL in West2East is related both to rapid prototyping, since West2East offers facilities for automatically generating Prolog code starting from the protocol specification, and to verification of interaction protocols, since it exploits Prolog meta-programming facilities for reasoning on the protocol itself in order to check whether some given condition on the protocol holds.

The paper is organised as follows: Section 2 provides an overview of West2East, while Section 3 explains why and how we use AUML for representing AIPs. Sections 4 to 6 describe the translation, code generation, and reasoning functionalities offered by West2East, respectively. An example of MAS engineered using West2East is discussed in Section 7, whereas Section 8 deals with related work and concludes. The reader is assumed to have some basic knowledge of WSDL, WS-BPEL, and Prolog.

2 The West2East toolkit

West2East is a prototypical Computer-Aided AOSE tool. It is available to the research community⁵, and provides a set of functionalities to the MAS developer aimed at making the engineering and implementation of interaction within a MAS faster, easier, and less error prone than if performed entirely by hand. In particular, West2East offers libraries for the automatic translation of AIPs represented in AUML into a) a textual notation proposed by Winikoff; b) an XML notation proposed by ourselves; c) a couple of WS-BPEL and WSDL documents; and d) a Prolog term (Figure 1). It also offers e) a mechanism for allowing agents that read a published AIP to reason about it before

³ http://centria.di.fct.unl.pt/~clima/
⁴ http://staff.science.uva.nl/~ulle/DALT-2006/home.html
⁵ http://www.disi.unige.it/person/MascardiV/Software/WEST2EAST.html
engaging in a dialog with its publisher, and f) libraries for the automatic generation of an executable program compliant to the original interaction protocol (Figure 2).

![Fig. 1. Translation functionalities of West2East](image1)

![Fig. 2. Reasoning and execution functionalities of West2East](image2)

The MAS scenario that we address is inspired by the provider/consumer model described by Luck et al. (2005): “an agent-oriented view of web services is gaining increased traction and exposure, since provider and consumer web services environments are naturally seen as a form of agent-based system.” We extend this point of view since we have provider and consumer agents instead of WSs, the good that is provided/consumed is a service that may be accessed by following a given AIP, and the AIP may be represented either using WS-BPEL and WSDL, like in any standard application based on WSs, or using other languages that might be preferred by the agents’ developers in specific domains or situations. Thus, our MASs are composed by agents that publish AIPs, named publishers or providers, and agents that read AIPs, named readers or consumers. Our reference scenario is an open system: the languages in which agents are implemented may be different, as well as their internal architecture and behaviour. The only features that two agents need to share for being able to interact are
the ability to understand the languages in which the AIP is represented (if these languages are WSDL and WS-BPEL, this requirement may be easily met since they are based on XML), and the parameters that must be shared in any communication between agents, namely the agent communication language, the message content language, and the ontologies to which messages refer.

Engineering the publisher agent requires that the MAS developer follows a set of steps leading to the description of the AUML AIP the publisher wants to be respected for offering its services. There are many AOSE methodologies that show how to design an AIP in AUML starting from the analysis of the system: Gaia, Zambonelli et al. (2003); Cernuzzi and Zambonelli (2004), INGENIAS, Gómez-Sanz and Pavón (2005), Prometheus, Padgham and Winikoff (2002), ROADMAP, Juan et al. (2002), and Tropos, Bresciani et al. (2004), are just some of them. We assume that the publisher’s developer takes advantage of one of the existing methodologies and tools for engineering the AIP in the correct way; West2East comes into play at this point. In fact, it can be used to generate, from the AUML AIP, the corresponding representation in one of the four notations introduced above, and discussed in detail in Section 4. In order to be accessed by the reader agent, the resulting document must be published in some ad-hoc repository. Besides for generating the document representing the AIP, West2East may be also used for automatically generating the code that corresponds to the AIP itself (Section 5).

As far as the reader is concerned, it reads the AIP document published by the provider, in order to “learn” the communication protocol to follow for obtaining the service. In this case, West2East can be used both for generating the executable code corresponding to the AIP, starting from the document retrieved from the repository, and for reasoning about it and deciding if and when it is appropriate to engage in a dialog with the publisher, as discussed in Section 6.

In order to use West2East, the MAS developer must be aware of the following features:

– First, any AIP must involve only two agents, the publisher and the reader. A publisher may provide and publish as many services as it wants, and a reader may interact with as many publishers as it wants. Besides this, the agent that plays the reader role in AIP $P_1$ may play the publisher role in AIP $P_2$, and vice versa. In fact, the pieces of code that are automatically generated for each AIP and for each role do not interfere with the ones generated for other AIPs and other roles. This makes the MASs that we are able to engineer similar to provider/consumer applications, where both the publisher-provider and the reader-consumer may interact with more than one consumer (resp. provider) following different interaction protocols. However, an agent can neither be both a provider and a consumer within the same AIP, nor a consumer (resp. provider) for more than one provider (resp. consumer) within the same AIP. Also, interleaved execution of two or more AIPs is not allowed. We are currently working for overcoming these limitations.

– Second, West2East can generate code in an automatic way starting from the AUML specification, from the XML one, from the Prolog one, and from the WSDL and WS-BPEL one, but not from documents in Winikoff’s textual notation. The implementation of a bidirectional translation facility from any supported notation to any other, that would eliminate this limitation, is under way.
Third, the generated code is written in Prolog, and is ready to be run inside the JADE agent platform. Although it can be executed as it is (in which case, all the boolean guards present in the AIP are assumed to be satisfied, as better explained in Section 5), it would be much more useful for the MAS developer to edit it, manually insert a belief base (namely, a set of Prolog atoms) representing the initial mental state of the agent, and manually add the AIP boolean guards referring to the agent’s mental state, inside the generated code. This requires basic skills in Prolog programming that are seldom found in practitioners from the industry. A similar consideration holds for the reasoning stage, that is performed exploiting meta-programming facilities offered by Prolog and that requires to express the conditions upon which performing the reasoning, as Prolog terms. We are working both for making the underlying Prolog code as transparent to the MAS developer as possible, by means of an intuitive graphical interface, and for generating code in languages other than Prolog, first of all Java.

By means of all the envisaged improvements, we hope to make West2East a tool useful not only for academic research, but also for industrial implementation of real MASs.

3 Visual representation of AIPs in AUML

We consider the AUML interaction diagrams proposed by the FIPA Modeling Technical Committee (2003) as our reference visual notation for AIPs.

As it is well known, AUML is a semi-formal graphical notation inspired by the UML standard, but aimed at not being restricted by UML. AUML, even if neither fully specified nor standardised, is widely adopted in the AOSE community: many AOSE methodologies use it as the basic notation technique, and some of the diagrams it provides are embedded in open source academic AOSE applications.

Besides the advantages that motivate its success, AUML also presents some shortcomings both from the semantic and the syntactic perspectives that have been partly overcome by recent notations (Sturm et al. 2003; Cervenka et al. 2005; Wagner 2004). Most new notations take inspiration from AUML, but build on more solid technical foundations and are better specified and documented. Nevertheless, we take AUML as our reference notation because AUML AIPs are expressive enough to let us represent the most common interaction activities, and, although AUML semantics is given in natural language, it can be easily formalised as discussed in Cabac and Moldt (2004). Also, AUML AIPs can be drawn using most of the existing UML-based CASE tools and can thus be exported in XMI, Object Management Group (2005a); we exploit XMI for moving from AUML to the textual notations supported by West2East. Using tools for UML editing is possible because AIPs in AUML are based upon the same set of visual symbols used in UML 2.0 sequence diagrams, Object Management Group (2005b), even if some symbols take a different meaning in AUML.

To make an example of an AIP in AUML, Figure 3 describes a FruitMarket scenario where a fruit seller (the agent on the left, named fs, playing the role of fruitSeller and belonging to the fruitSellerClass) interacts with a fruit buyer (the agent on the right, named fb, playing the role of fruitBuyer and belonging to the fruitBuyerClass). The content language is first order logic, and we
assume that an ontology named FruitMarketOntology contains all the information required by the agents to understand the content of the exchanged messages. Since it is becoming the de facto standard, we adhere to the FIPA-ACL message syntax, FIPA Technical Committee (2002); as a consequence, the ACL parameter always assumes the “FIPA-ACL” value in our diagrams, and we drop it from the diagram for readability.

The message that fires the protocol is a request from the buyer to the seller about the availability and price of a certain kind of fruit, represented by the logical variable $F$. According to the availability of the requested fruit, the seller can

- either perform a sequence of activities (**weak sequencing box**) that consist in first informing the buyer that fruit $F$ is available, and then proposing to buy the fruit at price $\text{EuroForKg}$. At this point, the seller may expect two reactions from the buyer (**alternative box**):
  - the buyer prefers to decide the delivery and payment mode, rather than getting a discount. Thus, it sends two requests about the accepted delivery modes and payment methods, to which the seller answers with two lists of possibilities, and then the buyer either accepts the proposal, choosing one delivery method and one payment method among the possible ones, or refuses.
  - the buyer prefers to get a discount, also if this means that it will not be able to choose the delivery and payment methods. Thus, it sends a call for proposal (c$\phi$p) to the seller, asking to make another proposal with discounted price, and fixed delivery and payment methods. The seller makes the proposal, and again, the buyer can either accept or refuse it.
- or it can inform the buyer that fruit $F$ is not available, and, optionally (**option box**), inform the buyer that another fruit $F_1$ is available instead.

## 4 Translation functionalities

Since visual diagrams like AUML AIPs have been conceived and designed for facilitating human beings, but are not suitable for being worked on by machines, we translate them into four different notations amenable for automatic processing. The “translation library” has been developed in Java. It takes in input the XMI description and, by exploiting the Extensible Stylesheet Language Transformations (XSLT) technology, W3C Working Group (1999), extracts the relevant features of the represented AIP from it. These features are used to generate a document in our XML notation, and, from it, documents in Winikoff’s textual notation, in WS-BPEL and WSDL, and in Prolog. The library also allows to translate the couple of WS-BPEL and WSDL documents into a Prolog term. This $\text{WS-BPEL + WSDL} \rightarrow \text{Prolog}$ translation takes advantage of JDOM6. We are currently extending the library in order to allow a bidirectional translation from any notation to any other among the supported ones, in order to reach the situation represented in Figure 4 (dotted arrows are translation links not yet implemented). This will allow agents to easily switch from any notation to any other thus making interaction among them easier, and to exploit the code generation and reasoning functionalities offered by West2East also when the AIP is represented in Winikoff’s notation.

4.1 Winikoff’s textual notation

The textual notation proposed by Winikoff (2005) has a significant scientific value per se as one of the first textual notations for AUML together with the one proposed by Koning and Romero-Hernandez (2003). Besides this, it is integrated in the Prometheus Design Tool (PDT\(^7\)) which offers the means for generating visual AUML diagrams from textual AIPs, thus complementing the functionalities offered by West2East.

PDT also supports code generation in JACK\(^8\), although at the moment this functionality does not look at the AIPs. PDT generates code for each agent, each capability, each plan, each event, and each data. It also generates events corresponding to percepts, and inserts some information corresponding to actions. Although this extension is not envisaged for the near future, AIPs might be used to help derive plans, so they would feed into code generation indirectly. In this case, West2East and PDT might be used in cascade - with Winikoff’s textual notation as the bridge between them - for generating JACK code compliant to an AUML diagram edited using any editor for UML 2.0.

Winikoff defines a textual AUML protocol as a sequence of commands (one per line). The first line defines the name of the protocol (\textit{start name}) and the last concludes the protocol (\textit{finish}). Commands in between are used to define agents (\textit{agent shortname longname}), messages (\textit{message}), start and end of boxes (\textit{box} and \textit{end}), types of the box (\textit{weak sequencing}, \textit{alternative}, \textit{option}, etc), boundaries between regions within a box (\textit{next}), guards (\textit{guard}), continuations (\textit{goto} and \textit{label}), and the end of an agent’s participation in the protocol (\textit{stop}).

We extend this textual notation with

– mandatory parameters (\textit{content-language} and \textit{ontology})
– optional parameters (to be eventually defined by the developer)
– information on the agent \textit{class} and \textit{role} (the short name is always \textit{p} for the publisher and \textit{r} for the reader, while the name is extracted by the AUML diagram)
– optional content of the messages

A portion of Winikoff’s representation of the protocol depicted in Figure 3 is shown below.

\begin{verbatim}
start sd FruitMarket FruitMarketOntology first_order_logic
agent-publisher p fs fruitSeller fruitSellerClass
agent-reader r fb fruitBuyer fruitBuyerClass
message r p request availability and price(fruit(F))
    box alternative
        box seq
            message p r inform available(fruit(F))
        box alternative
            message r p request delivery modes
            message p r inform delivery mode(ListOfModes)
            message r p request accepted payment methods
            message p r inform accepted payment methods(ListOfMethods)
    ...
\end{verbatim}

\(^7\)http://www.cs.rmit.edu.au/agents/pdt/
\(^8\)http://www.agent-software.com
4.2 Our XML notation

The advantages of using XML to exchange information are widely recognised. XML is highly interoperable, easy to read both for human beings and for machines, and many tools embed parsers for it. West2East offers the functionality to output an XML representation of the AIP that corresponds to Winikoff’s one, but has the advantage of not requiring any ad-hoc parser. A portion of the protocol depicted in Figure 3 is shown below. We think this notation, as Winikoff’s one, is self-explanatory, thus we do not further comment on it.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<protocol-digram-intermediate-notation xmlns:UML2="org.omg.xmi.namespace.UML2"
xmlns:UML="org.omg.xmi.namespace.UML">
<parameters>
<ontology>"FruitMarketOntology"
<CL>"first order logic"
</parameters>
<protocol-name>sd FruitMarket</protocol-name>
<agent-publisher short-name="p">fs/fruitSeller:fruitSellerClass</agent-publisher>
<agent-reader short-name="r">fb/fruitBuyer:fruitBuyerClass</agent-reader>
<main-interaction-fragment>
<combined-fragment>
<type>seq</type>
<guard/>
<fragment>
<message>
<sender>reader</sender>
<communicative-act>request</communicative-act>
<content>availability and price(fruit(F))</content>
</message>
</fragment>
...

4.3 WSDL and WS-BPEL

We have already anticipated in Section 1 the need and the usefulness of a bridge between AOSE methodologies and notations, and the WS infrastructure. It is common opinion that both parties involved would benefit from this bridge, and that the advantage of an integrated “AOSE + WSs” approach would be greater than the sum of the advantages of the two approaches taken in isolation. These considerations motivate the third translation functionality offered by West2East, namely the one whose target is a couple of WSDL and WS-BPEL documents. We assume that the reader has some familiarity with these two languages. Many introductory resources about them can be found on the Web\(^9\).

The suitability of WS-BPEL for representing AUML AIPs is supported by the close relationships between the constructs offered by WS-BPEL and those offered by AUML, summarised in Figure 5. AUML is an easily comprehensible visual language that allows to express the high level structure of interaction protocols, while WS-BPEL is a quite

\(^9\) Consider for example the two tutorials [http://www.w3schools.com/wsdl/default.asp](http://www.w3schools.com/wsdl/default.asp) and [http://www.eclipse.org/stp/b2j/docs/tutorials/wsbpel/wsbpel_tut.php](http://www.eclipse.org/stp/b2j/docs/tutorials/wsbpel/wsbpel_tut.php), dealing with WSDL and WS-BPEL, respectively.
complex XML language suitable for expressing not only the high level structure of business processes involving a set of WSs, and consequently the interactions between them, but also details of message exchange, operations requested to the WSs, and guard conditions. WS-BPEL is thus powerful enough to represent all the information contained in AUML diagrams.

To make an example, the AUML protocol depicted in Figure 3 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"
9: myRole="seller" partnerRole="buyer"/>
4: <partnerLink name="readerPL" partnerLinkType="lns:BuyerSeller"
8: myRole="buyer" partnerRole="seller"/>
5: </partnerLinks>
6: <variables>
7: <variable name="continue1" element="lns:continue1" type/>
8: <variable name="choose2" element="lns:choose2" type/>
9: ....
13: </variables>
14: <copy><from opaque="yes"/><to>$continue1.value</to></copy>
15: <while condition="$continue1.value=true">
16: <sequence>
17: <receive partnerLink="publisherPL" portType="lns:publisherPT"
6: operation="RCV_Mess1" createInstance="yes"/>
18: <copy><from opaque="yes"/><to>$choose2.value</to></copy>
19: <switch>
20: <case condition="$choose2.value=1">
21: <if condition="$condition_6.value=true"/>
22: <invoke partnerLink="publisherPL" portType="lns:publisherPT"
7: operation="SND_Mess15"/>
23: <receive partnerLink="readerPL" portType="lns:readerPT"
5: operation="RCV_Mess15"/>
24: </sequence>
25: </case>
26: </switch>
27: </sequence>
28: <copy><from opaque="yes"/><to>$continue1.value</to></copy>
29: </while>
30: </process>
```

**WSDL specification**

```
1: <message name="Mess_15" type="xs:string"
3: fixed="i_may_sell(fruit(F1))"/>
4: ....
5: <part name="performative" element="inform_ca"/>
6: <part name="Participant" type="tns:Participant_MsgFromPublisher"/>
7: <part name="Content" element="tns:content_of_Mess_15"/>
8: <part name="Content.Language" element="tns:content_language_name"/>
9: <part name="Ontology" element="tns:ontology_name"/>
10: <part name="Protocol" element="tns:protocol_name"/>
11: </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 (used by internal/private functions, as opposed to transparent data relevant to public aspects) 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 in  
an 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 exe-  
cution of that protocol branch (condition = "$continue_1.value = true", for  
example).

Apart from the first message of the protocol, that, according to the WS-BPEL spec-  
ification, must be received by the publisher of the document, the points of view of both  
the publisher and the reader are taken into account when describing communicative ac-  
tions. 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 view-  
point, and from the reader’s one. The WSDL document describes the details of each  
exchanged message: for example message 15 has an inform performative (line k+1)  
and i.may_sell(fruit(F1)) content (line j).

4.4 Prolog

The last translation functionality offered by West2East goes from AUML to Prolog. The  
choice of this language was driven by the code generation and reasoning functionali-  	ies that, for the moment, are implemented in Prolog and thus were easier to program  
starting from a Prolog term rather than from any other notation. Another reason was  
the possibility to use a Prolog-based model checker like LMC\textsuperscript{10} to verify properties of  
the AIP, using techniques alternative to those described in Section 6. We have not yet  
started to experiment LMC, but we trust that adapting our Prolog notation to it will  
require a small effort.

To give the flavor of the Prolog notation, a fragment of the FruitMarket AIP is in-  
cluded below. We think that any reader with a basic knowledge of first-order logic can  
understand this representation, since it is just a syntactic variation of Winikoff’s and  
XML ones. The only difference that should be noted is the usage of JADE agent identi-  
fiers for the publisher’s and reader’s short names (‘seller@giocas:1099/JADE’  
and ‘buyer@giocas: 1099/JADE’ respectively). These identifiers must be added by  
hand by the MAS developer, if she wants to automatically generate the Prolog code of  
the two agents in such a way that they can be run (eventually without any modification)  
inside JADE.

\textsuperscript{10} http://www.cs.sunysb.edu/~lmc/
5 Automatic generation of the protocol-compliant code

Automatic generation of code from specifications allows the MAS engineer to save programming time and thus to concentrate on the engineering stages that require more brainpower. West2East offers a “Prolog-oriented” code generation functionality; the generation of a Prolog program starts from the Prolog representation of the AIP, and a finite state machine corresponding to the AIP is simulated by the generated clauses. States are meaningless terms only used to enforce the correct transitions, and transitions correspond either to communicative actions, or to check of conditions. Although the generated clauses have a very simple structure \( \text{exec(FromState) :- Transition, exec(ToState), where Transition may also be the null one, thus leading to even simpler clauses, exec(FromState) :- exec(ToState)}. \) some background on Prolog programming might help to face this and the next sections.\(^{11}\)

According to the nature of AIPs we consider, transitions can be of four kinds: send transition, receive transition, check transition and null transition. The initial and final fragments of the Prolog code corresponding to the protocol depicted in Figure 3 are shown below (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.

\(^{11}\) The reader may consider one of the on-line Prolog tutorials, such as http://www.coli.uni-saarland.de/~kris/learn-prolog-now/lpnpage.php?pageid=online, besides the classic book “The art of Prolog”. Sterling and Shapiro (1994).
clause 1:  
exec(s('sd FruitMarket',0)) :- 
    check_guards(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(s('sd FruitMarket',0),0),0),1)) :- 
    check_guards(s(s(s(s('sd FruitMarket',0),0),0),1), 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(s('sd FruitMarket',0),0),1),0),0),1)) :- 
    receive('PROPOSE','buy(fruit(F),price(Euro))','seller@giocas:1099/JADE'), 
    exec(s(s(s(s(s(s('sd FruitMarket',0),0),1),0),1),0)).

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

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

clause n-1:  
exec(s(s(s(s(s(s(s(s('sd FruitMarket',0),0),1),0),1),0),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 were specified by the original 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 conditions that depend on the agent’s belief base (also to be manually added to the generated code).

The predicate send in clause 5 (resp., receive, in clauses 8, 9 and n-1) characterises a send transition (resp., a receive transition). These communication predicates are implemented by the DCaseLP library, Gungui et al. (2005), that provides an interface between the tuProlog implementation of a Prolog interpreter, Denti et al. (2005), and the communication facilities offered by JADE.

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 structured 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 \texttt{exec(ProtocolId, final)} always succeeds (clause n).

– **Translating cycles.** A \texttt{while(Guard, WhileActivities)} action performed in the state \texttt{s(S,I)} for reaching the state \texttt{SFinal}, is translated into

\begin{verbatim}
class a: exec(s(S,I)) :- check.guard(s(S,I),Guard), exec(s(s(S,I),0)).
class b: exec(s(S,I)) :- exec(SFinal).
class c: exec(s(S,I)) :- exec(s(S,I)).
\end{verbatim}

Clauses that translate the \texttt{WhileActivities} from \texttt{s(s(S,I),0)} to \texttt{s(S,I)}

Our fruit market AIP starts with a \texttt{while} activity performed in the state \texttt{s(‘sd FruitMarket’,0)} for reaching the state \texttt{s(‘sd FruitMarket’,final)}. Clause 0 of our code fragment corresponds to the first clause of the translation of the \texttt{WhileActivities} (clause a); clause 2 corresponds to clause b; and clause 3 to clause c. All the remaining activities of the protocol correspond to the \texttt{WhileActivities}, and they will need to end with reaching state \texttt{s(‘sd FruitMarket’,1)} (corresponding to \texttt{s(S,I)} in clause c). When discussing the translation of options, we will see that this truly happens.

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

\begin{verbatim}
class a: exec(s(S,I)) :- communication(Perform,Cont,Addr), exec(SFinal).
\end{verbatim}

Examples of this translation are clauses 5, 8, 9, and \texttt{n-1}.

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

\begin{verbatim}
class a: exec(s(S,I)) :- exec(s(s(S,I),0)).
Class that translates Activity0 from s(s(S,I),0) to s(s(S,I),1)
Class that translates Activity1 from s(s(S,I),1) to s(s(S,I),2)
\ldots

Class that translates ActivityN from s(s(S,I),N) to SFinal
\end{verbatim}

An example of this translation are clauses 4 and 5. The state \texttt{s(S, I)} from which the translation starts is \texttt{s(‘sd FruitMarket’,0)} (\texttt{S = s(‘sd FruitMarket’,0); I = 0}) and the state to reach is \texttt{s(‘sd FruitMarket’,1)}. Clause 4 corresponds to clause a, while clause 5 corresponds to the translation of the first activity within the sequence, \texttt{send(‘REQUEST’, ‘availability and price(fruit(F))’, ‘seller@giocas:10-99/JADE’)} from \texttt{s(s(S,I),0)} to \texttt{s(s(S,I),1)}. Clause \texttt{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 \texttt{switch([case(Guard0,Activity0), ..., case(GuardN,ActivityN)])} action performed in the state \texttt{s(S,I)} for reaching the state \texttt{SFinal}, is translated into

\begin{verbatim}
class a: exec(s(S,I)) :- check.guard(s(s(S,I),0),Guard0), exec(s(s(S,I),0)).
Classes that translates Activity0 from s(s(S,I),0) to SFinal
\ldots

class z: exec(s(S,I)) :- check.guard(s(s(S,I),N),GuardN), exec(s(s(S,I),N)).
Classes that translates ActivityN from s(s(S,I),N) to SFinal
\end{verbatim}

Clause 6 is an example of translation of a \texttt{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 \texttt{exec(s(s(‘sd FruitMarket’,0),}...
as clause 6, not shown in the program fragment, corresponds to clause 3, 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 \( S_{\text{Final}} \), is translated into

\[
\begin{align*}
\text{clause a:} & \quad \text{exec}(s(S, I)) :- \text{check.guard}(s(S, I), \text{Guard}), \text{exec}(s(s(S, I), 0)). \\
\text{clause b:} & \quad \text{exec}(s(S, I)) :- \text{exec}(S_{\text{Final}}).
\end{align*}
\]

Clauses \( m-3 \) and \( m-2 \) correspond to clauses \( a \) and \( b \) respectively, where \( s(S, I) \) corresponds to \( s(s(s(s(s(\text{sd FruitMarket}',0),0),1),1),1) \) and \( S_{\text{Final}} \) to \( s(\text{sd FruitMarket'},1) \). ThenActivities correspond to the reception of the message ‘\( i \_\text{may\_sell}(\text{fruit}(F1)) \)’, after which the agent moves to \( s(\text{sd FruitMarket'},1) \) (clause \( m-1 \)), as anticipated when we introduced the translation of cycles.

Both the publisher and the reader may take advantage of the automatic code generation functionality. In fact, given the same Prolog AIP (that can be obtained directly from the AUML AIP by the publisher, and from any textual representation of the AIP but Winikoff’s one by the reader), the only difference between the code generated by the publisher and that generated by the reader is a swap of senders and receivers inside exchanged messages. The West2East library for generating code has the agent role (either publisher or reader) among its parameters, and thus it can produce the right code using the same algorithm.

6 Reasoning about the protocol

The reasoning functionality offered by West2East can be exploited by both the publisher and the reader, but it makes sense only for the latter one, since the publisher is expected to publish an AIP designed to meet its requirements.

In order to perform the reasoning stage, the Prolog program generated by West2East must contain the fact \( \text{condition\_to\_check}(\text{Condition}) \). \( \text{Condition} \) must be instantiated with \( \exists(\text{Action}), \forall(\text{Action}), \text{or no\_cond} \) (the default for this fact, meaning that no checks on the protocol are made). \( \text{Action} \) may correspond to one of the transition types, send, receive, check. The reader agent may thus check that there is one possible path where (resp. in any possible path) a send(Performative, Content, Receiver), or a receive(Performative, Content, Sender), or a check_guard(State, Guard), is executed. If the condition is satisfied, the reader agent engages in a dialog with the publisher trying to enforce the execution of the “right” path, as long as the choice is up to it.\(^\text{12}\)

To go on with our fruit market example, let us consider a “restrictive” fruit buyer agent that accepts to interact with the fruit seller agent only if it provides - whatever the choice is up to it.

\(^{12}\) In Prolog technical words, the reasoning stage is implemented by allowing the agent to make a depth-first exploration of the SLD-tree for \( P \cup \{G\} \) via \( R \), where \( P \) is the program generated as described in Section 5, \( G \) is the goal that starts the execution of the protocol \( \text{exec}(s(\text{sd FruitMarket'},0)) \) in our example, and \( R \) is the leftmost selection rule. This exploration is aimed at either finding one path where the desired condition is verified (for demonstrating that an existential property holds), or finding a path where the final state is reached, and the desired condition is not verified (for demonstrating that a universal property does not hold).
protocol branch followed - a bunch of payment methods among which the buyer can choose. The fruit buyer agent’s code should contain the fact: condition_to_check(forall(receive('INFORM', 'accepted_payment_methods(ListOfMethods)', Sender))). This condition is not verified by the protocol depicted in Figure 3, as it can be easily seen. Thus, the “restrictive” fruit buyer agent would 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 action 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 buyer agent would start the protocol execution.

While executing, the buyer explains its actions by printing them on a log file:

```plaintext
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('sd FruitMarket', 0), 0), 1), no guard)
receive('PROPOSE', 'buy(fruit(F), price(EuroForKg))', 'seller@giocas:1099/JADE')
check guard(s(s(s(s('sd FruitMarket', 0), 0), 1), 0), 2), 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')

I executed the statement check guard(s('sd FruitMarket', 0), no guard)
```

Nondeterministic action: I received the message I was waiting for, namely

```plaintext
('INFORM', 'available(fruit(F))', 'seller@giocas:1099/JADE')
```

Nondeterministic action: I received the message I was waiting for, namely

```plaintext
('INFORM', 'delivery mode(ListOfModes)', 'seller@giocas:1099/JADE')
```

I executed the statement

```plaintext
send('REQUEST', accepted payment methods, 'seller@giocas:1099/JADE')
```

Nondeterministic action: I received the message I was waiting for, namely

```plaintext
('INFORM', 'accepted payment methods(ListOfMethods)', 'seller@giocas:1099/JADE')
```

I have reached my goal!

7 Engineering and Implementing a MAS exploiting West2East

In this section we discuss a possible usage of West2East for implementing a MAS. The textual representation of AIPs that we have chosen for this example is WSDL + WS-BPEL, and the infrastructure for running the MAS is JADE. An approach similar to that discussed in this section might be followed for implementing JADE agents that represent AIPs in any other notation among the supported ones, as well as for "non-
JADE” agents, still implemented in Prolog, that use the infrastructure already available for WSs for advertising and exchanging services.

With this respect, it is interesting to note that many proposals for frameworks that integrate WSs and JADE can be found in the literature, for example Greenwood and Calisti (2004); Agentcities Task Force (2003); Thang and Kowalczyk (2005). Although West2East may prove useful for engineering agent systems based both on WSs and on JADE (and also JADE agent systems that exploit languages for WSs, as in this example), it is independent from any specific infrastructure.

To implement our publisher/reader MAS, we have developed a JADE Protocol Manager Agent (PMA) that, together with JADE’s Directory Facilitator (DF), allows agents to publish and retrieve WSDL + WS-BPEL specifications representing AIPs.

The development of the publisher agent must take the following activities into account (see Figure 6):

- p.0 design of an AIP in AUML following an AOSE methodology (outside the scope of West2East);
- p.1 translation from the AUML visual diagram to the WSDL + WS-BPEL notation discussed in Section 4.3;
- p.2 advertisement of the WSDL + WS-BPEL document by exploiting the services offered by the DF and the PMA, as described later in this section;
- p.3 translation from the AUML visual diagram to the Prolog notation discussed in Section 4.4;
- p.4 generation of the Prolog code from the Prolog term representing the AIP, as described in Section 5;
- p.5 execution of the Prolog code in JADE extended with the DCaseLP libraries.

The reader agent must be engineered in order to follow the steps shown in Figure 7:

- r.1 interrogation of JADE’s DF and access to the repository managed by the PMA in order to retrieve the WSDL + WS-BPEL documents;
- r.2 translation from the WSDL + WS-BPEL notation to the Prolog one;
- r.3 generation of the Prolog code from the Prolog term;
- r.4 reasoning stage discussed in Section 6;
- r.5 execution of the Prolog code in JADE extended with the DCaseLP libraries.

For supporting the interaction between the publisher and the reader, we have designed and implemented the MAS depicted in Figure 8. We will name “Agent Services” both the general management services offered by JADE’s DF and by the PMA that we implemented, and the application specific services such as the fruit-selling service offered by the fruit seller and advertised by publishing the WSDL + WS-BPEL document discussed in Section 4.3.

The DF is provided by JADE, and offers a “yellow pages” service allowing agents to publish agent services, so that other agents can find and exploit them. An agent wishing to publish a service must send information about itself and about the service it provides, to the DF. The DF is used in conjunction with the PMA that offers a service for the advertisement and retrieval of AIPs’ specifications, and stores the WSDL + WS-BPEL documents on a MySql DBMS13.

13 http://www.mysql.com/
When the JADE platform is started, the PMA registers the protocol-publishing and the protocol-reading agent services to the DF (arrows 1 and 2 in Figure 8). When a publisher agent wants to advertise an agent service offered by means of an AIP initially specified in AUML, it
1. generates the WS-BPEL + WSDL documents from the AUML AIP;
2. looks in the DF to find the protocol-publishing agent service (arrows 3, 4);
3. sends a message to the PMA with the WSDL + WS-BPEL document, and waits to receive the protocol identifier (PID) assigned by the PMA to it (arrows 5, 6);
4. registers the agent service 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 agent service properties.

When a reader agent looks for an agent service, it
1. queries the DF to find the required agent service (arrows 9, 10) obtaining the name and address of the publisher, the address of the PMA, and the PID that the PMA assigned to the agent service (if they exist in the DF);
2. sends a message to the PMA to obtain the WSDL + WS-BPEL document representing the AIP (arrows 11, 12) and identified by the PID obtained in 1;
3. from the WSDL + WS-BPEL specification, generates the corresponding Prolog term, and, from this, the Prolog code, and reasons about it;
4. according to the reasoning outcome, eventually starts the protocol-compliant communication in order to obtain the agent service (direct communication between the publisher and the reader, represented by dashed arrows in Figure 8).

Figure 9 refers to an execution run of the MAS composed by one PMA, one reader, one publisher, and the DF. In this figure, corresponding to the example of Figure 3, the publisher plays the role of fruitSeller while the reader plays the role of fruitBuyer. The first twelve messages correspond to the communication represented by the twelve solid arrows in Figure 8, while the other messages are exchanged during the execution of the fruitMarket AIP, aimed at allowing the fruitBuyer to obtain the fruit-selling agent service 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.

8 Related work and conclusions

The related work that we consider in this section deals with the three features that better characterise West2East: the exploitation of (A)UML for engineering WSs, the automatic generation of code starting from the AIP specification, and the capability of reasoning about AIPs.

As far as the first feature is concerned, the literature describes many proposals of using UML for modelling WSs. For example, Grønmo and Solheim (2004) and Bauer and Huget (2004) share the same purpose of exploiting UML for modelling WS composition. Both of them propose extensions to UML, when it does not provide enough support for the given purpose: those suggested by Grønmo and Solheim are within a composite WS model that represents a gas dispersion emergency case, but can be generalised to other domains, while those proposed by Bauer and Huget are agent-oriented ones. The
paper by Provost (2003) discusses how to design WSs in UML, express their semantics in WSDL, and implement them in a supporting programming language. While no tools for generating the WSs specification from the UML one are described in these three papers, Gardner (2003) describes a UML profile for automated business processes which allows BPEL4WS \(^{14}\) processes to be modelled using an existing UML tool and automatically mapped to BPEL4WS, and Mantell (2005) describes a tool which takes processes defined by UML activity diagrams and generates the corresponding BPEL4WS and WSDL files. The key difference between our work and both Gardner’s and Mantell’s ones is that we translate interaction protocols represented as AUML protocol diagrams (that are an extension of UML sequence diagram) into WS-BPEL abstract processes, while they start from UML activity diagrams and generate executable BPEL4WS processes. The tool for automatic generation referenced by both authors is the same, namely a plug-in integrated inside the Emerging Technologies Toolkit by IBM\(^ {15}\), and no longer available.

The generation of executable code from an AUML specification has been dealt with by some papers and implemented toolkits, described for example in Dinkloh and Nimis (2003); Huget (2002); Gomez-Sanz and Pavon (2003); Rooney et al. (2004); Cabac and Moldt (2004), while the generation of code from WS-BPEL and UML 2.0 specifications is already supported by many commercial tools. However, West2East purpose goes further beyond an \( (A)UML \rightarrow \text{code} \) or a \( \text{WS-BPEL} \rightarrow \text{code} \) translation, since it aims at making the MAS analysis and design, carried out following AOSE methodologies, closer to its implementation and deployment. West2East offers functionalities for generating code also from textual representations of the AIP (different from UML or AUML), and can be used for generating a document in one of these textual notations starting from the visual AUML diagram, without necessarily generating the corresponding code. One possible usage of the \( AUML \rightarrow \text{textual notations} \) functionality has been discussed in Section 4.1, where we envisaged that, for example, West2East might be used in conjunction with PDT for generating JACK code from AUML visual interaction diagrams, using Winikoff’s notation as a bridge. Sections 4.2 to 4.4 highlight other possible usages and advantages.

Finally, the Prolog-related functionalities offer a suitable way to realise MASs in which new agents are able to learn how to interact with existing agents by reasoning about AIPs. These functionalities may be exploited for reasoning about AIPs in general, but can also be used to reason about WSs, in particular. The need of reasoning about WSs is extremely pressing, since the full exploitation of the WS technology requires to address issues of security and privacy protection, that West2East reasoning mechanism, although simple, may help to cope. We represent AIPs using languages for WS orchestration, namely WSDL and WS-BPEL, thus complementing many proposals that can be found in the literature where AIPs are described by means of choreography languages. For example, Baldoni et al. (2005) propose an approach to the verification of a priori conformance of a business process to a protocol, which is based on the theory of formal languages and guarantees the interoperability of agents that are individually proved to conform to the protocol. Brogi et al. (2004) formalise the WSCI language.

\(^{14}\) BPEL4WS is the previous version of WS-BPEL.

\(^{15}\) http://www.alphaworks.ibm.com/tech/ettkws
W3C Web Services Choreography Working Group (2002), and discuss the benefits that can be obtained by such formalisation, in particular the ability to check whether two or more WSs are compatible to inter operate or not, and, if not, whether the specification of adaptors that mediate between them can be automatically generated. A unifying view of orchestration and choreography is suggested by Busi et al. (2005) that define a notion of conformance between choreography and orchestration which allows to state when an orchestrated system conforms to a given choreography. Choreography and orchestration are formalised by using two process algebras and conformance takes the form of a bisimulation-like relation.

The originality of West2East goes beyond the originality of each single functionality it provides, and lies in the capability of offering to the MAS developer a bunch of choices integrated within the same environment. The contribution of West2East to the acceptance of AOSE approaches outside the boundaries of academic research may be found in its effort to

– promote the use of AOSE methodologies in the realisation of commercial MASs by offering a set of functionalities that support the MAS implementation starting from previous analysis and design stages that led to the definition of AIPs in AUML;
– ease the development of agents thanks both to the automatic translation of AIPs into Prolog code, and to the direct support of agent implementation inside a well-accepted platform for agent-based software development, JADE;
– facilitate the interaction between existing and new agents thanks to the representation of AIPs in a widely used and standardised language, WS-BPEL.

The two main directions of our work involve the extension of the set of notations and languages that West2East supports, both for representing AIPs and for implementing agents, and its evaluation on some application larger than those developed, by us, since now. For facing the last activity, we plan to involve the bachelor and master students of the Artificial Intelligence course that we teach. Given the same MAS to build, some of them will be asked to follow an existing AOSE methodology for designing the AUML AIP, and to use West2East for implementing a MAS in JADE, while some others will be asked to build the MAS directly in JADE. By comparing the average performance of the two groups of students, we will be able to measure time-saving and reduction in error gained by using West2East in conjunction with a principled AOSE approach.

Acknowledgments

This work was partially supported by the Italian project MIUR PRIN 2005 “Specification and verification of agent interaction protocols”.
Fig. 3. Fruit marketplace protocol in AUML
Fig. 4. Bidirectional translation between the different notations supported by West2East

<table>
<thead>
<tr>
<th>AUML</th>
<th>WS-BPEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Roles</td>
<td>ag-name/ ag-role: ag-class box</td>
</tr>
<tr>
<td>Message</td>
<td>myRole and partnerRole tags</td>
</tr>
<tr>
<td>Sequence</td>
<td>Unspecified</td>
</tr>
<tr>
<td>Condition</td>
<td>sequence</td>
</tr>
<tr>
<td>Option</td>
<td>Switch</td>
</tr>
<tr>
<td>Cycle</td>
<td>IF</td>
</tr>
<tr>
<td>Content</td>
<td>while</td>
</tr>
</tbody>
</table>

Fig. 5. Correspondence between AUML and WS-BPEL concepts

Fig. 6. Activities for the development of a publisher agent in JADE using West2East

Fig. 7. Activities for the development of a reader agent in JADE using West2East
Fig. 8. Our MAS implemented in JADE

Fig. 9. Execution run in JADE


