
West2East: exploiting WEB Service Technologies to Engineer Agent-based Software

Giovanni Casella

DMI, Università di Salerno
Via Ponte don Melillo
84084, Fisciano, Italy
E-mail: casella@disi.unige.it
and
DISI, Università di Genova
Via Dodecaneso 35, 16146, Genova, Italy

Viviana Mascardi*

DISI, 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 (AOSE) toolkit aimed at supporting the implementation of Multiagent Systems (MASs). West2East exploits languages for Web Services (WSs) for sharing Agent Interaction Protocols (AIPs), modelled using AUML, across the web, and Computational Logic to reason about them. In particular, West2East offers libraries for the translation of AIPs represented in AUML into 1) a textual notation proposed by Winikoff; 2) an XML notation proposed by ourselves; 3) a couple of WS-BPEL and WSDL documents; and 4) a Prolog term. It also offers 5) a mechanism for allowing agents that read a published AIP to reason about it before engaging in a dialogue with its publisher, and 6) libraries for the automatic generation of an executable program compliant with the original interaction protocol.

Keywords: intelligent agent; Web Service; WS; Computational Logic; CL; Agent Interaction Protocol; AIP; Agent-Oriented Software Engineering; AOSE.

Reference to this paper should be made as follows: Casella, G. and Mascardi, V. (2007) 'West2East: exploiting WEB Service Technologies to Engineer Agent-based Software', *Int. J. Agent-Oriented Software Engineering*, Vol. 1, Nos. 3/4, pp.396–434.

Biographical notes: Giovanni Casella is a PhD student in Computer Science at the University of Salerno, where he obtained his Master's degree. At the end of 2005, he moved to the Department of Computer Science, University of Genova, as a visiting student. His research interests include issues related to web services, intelligent agents and automatic recognition of hand-sketched diagrams. He has published papers in national journals and conference proceedings.

Viviana Mascardi received her PhD in Computer Science from the University of Genova. She is an Assistant Professor at the Department of Computer Science, University of Genova. Her research interests are related to agent-oriented software engineering, languages for programming agents, web services and ontologies. She has published research papers in national and international journals, conference proceedings and chapters of books.

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 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 system following the Multiagent System (MAS) metaphor, while Web Services (WSs) provide an already available and widely accepted infrastructure for supporting interoperable machine-to-machine interaction over a network. WSs allow 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’¹ and ‘Societies of Computees (SOCS)’², 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”.

This paper describes ‘West2East’, a tool that exploits ‘WEB Service Technologies to Engineer Agent-based SofTware’ starting from the specification of an Agent Interaction Protocol (AIP). West2East exploits AUML for representing AIPs; many different languages, including standard languages for WSs, for sharing them; and Computational Logic to reason about them. Although West2East is mainly aimed at developing MASs based on the WSs technological infrastructure, it is not limited to this. 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 also be feasible.

West2East consists of a set of libraries for:

- Translating visual AUML AIPs to various formats

Starting from an AUML interaction diagram graphically drawn using any UML editor, West2East generates the corresponding representation in:

- a textual notation proposed by Winikoff
- b XML notation proposed by ourselves

- c couple of WS-BPEL and WSDL documents
- d Prolog term.

- Generating a code compliant to the AIP

A Prolog program for each party involved in the AIP is automatically generated by West2East. After a manual completion for adding the information missing in the AIP's specification, such as agents' state and guards of conditions, the Prolog code can be run inside JADE thanks to the DcaseLP libraries (Gungui *et al.*, 2005), which allow the integration in JADE of tuProlog agents (Denti *et al.*, 2005).

- Reasoning about the AIP

A mechanism for allowing agents to reason about an AIP is provided by West2East. Existential and universal properties, such as 'There is one path of the protocol where I will receive *message₁*', and 'Whatever the path, I will send *message₂*', can be verified. This mechanism can be used by an agent that reads a published AIP and wants to reason about it before engaging in a dialogue with its publisher.

Thanks to the functionalities provided by its libraries, West2East supports software engineers in handling AIPs in the MAS's design and verification. Moreover, the tool offers facilities for building software agents able to interact with other agents in open and dynamic systems, where interaction protocols are not known in advance, and to reason about these interactions.

The scientific contributions that West2East brings to research in the AOSE field may be summarised as:

- offering an agent-centred methodological approach to the development of web applications
- offering an implemented Integrated Development Environment for supporting the agent-centred development.

The paper is organised as follows: Section 2 provides the background and the motivation for our research. Section 3 provides an overview of West2East, while Section 4 explains why and how we use AUML for representing AIPs. Sections 5 to 7 describe the format translation, code generation and reasoning functionalities offered by West2East, respectively. An example of a MAS engineered using West2East is discussed in Section 8, whereas Section 9 deals with related work. Finally, Section 10 highlights the future directions of our work and concludes. The reader is assumed to have some basic knowledge of WSDL, WS-BPEL and Prolog.

2 Background and motivation

West2East builds upon three technologies strictly related to each other: Intelligent Agents, WSs and Computational Logic (CL).

2.1 Agents and WSs

The literature in the agents' field devotes much space to exploring the relationships between agents and WSs. Walton (2005), Buhler and Vidal (2005) and Bozzo *et al.* (2005) observe that in the realisation of complex systems, WSs are the *infrastructure providers*, while agents are the *coordination framework* providers. 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 WS infrastructure. For this reason, we propose the use of AUML to model AIPs, and WS-BPEL (OASIS WSBPEL Technical Committees, 2006) plus WSDL (Web Services Description Working Group, 2006) to share their specification.

2.2 Agents and CL

As pointed out in Ciancarini and Wooldridge (2000), in the 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. 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 of 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 DALT.⁴ Various surveys and monographic collections on this topic are also available, such as Sadri and Toni (1999), Dix *et al.* (2003) and Mascardi *et al.* (2004).

2.3 CL and WSs

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 terms, may be fruitfully represented and verified adopting CL (Alberti *et al.*, 2006). CL is also suitable for dealing with WS composition (Rao *et al.*, 2004; McIlraith, 2004) and selection (Baldoni *et al.*, 2006).

West2East represents an effort to fill the gap between visual representations used to design AIPs by industrial practitioners (*i.e.*, UML and its extensions) and formal representations of AIPs required for reasoning about interaction in a formal and principled way. The ability of West2East to automatically generate a representation of the AIP in the standard WS-BPEL and WSDL formats, starting from an AUML AIP, may be exploited by any average-skilled software developer who knows how to draw an AUML interaction diagram and wants to share the AIP that it represents across the web using a standard and widespread notation, without needing to know WS-BPEL and WSDL. It may also be exploited by any agent-oriented software engineer who wants to import into the agent field research results applicable to WS-BPEL verification – such as tools and algorithms for analysing BPEL processes using Petri Nets (Verbeek and van der Aalst,

2005; Hinz *et al.*, 2005); graph-transformation-based approaches for checking the adherence of a WS-BPEL process to privacy policies (Li *et al.*, 2006) and tools for verifying properties of conversations generated by a composite web service (Bultan *et al.*, 2006).⁵

Besides creating WS-BPEL and WSDL documents, West2East also enables the generation of other textual notations usable by a machine to obtain a formal verifiable representation of the AIP. A software engineer who wants to verify some properties of an AIP may thus draw it as an AUML diagram using any UML editor, and exploit West2East to generate both the corresponding documents in various textual formats (which might be reworked in order to be input to a suitable model checker) and the corresponding executable code, amenable to being run and tested. Reworking an XML or a logic-based notation for being accepted as the input of some model checker is easier than adapting the graphical AUML visual diagram. The ‘XML → model checker notation’ approach has been successfully followed, for example, by Fu *et al.* (2004b), who translate bounded XML data and XPath expressions into Promela, the input language of SPIN (Holzmann, 2003). The techniques they use constitute the basis of the Web Service Analysis Tool (WSAT – Fu *et al.* (2004a)), which verifies linear time-logic properties of composite WSs. The logic-based notation output by West2East may be used, after an adaptation stage, as the input for logic-based model checkers such as XMC and XSB (Cui *et al.*, 1998).

By exploiting the features offered by CL, West2East also implements a simple but effective tool for the generation of working prototypes of agents that respect a given interaction protocol, through the automatic generation of the Prolog code starting from the AUML protocol specification, and for the verification of AIP properties, since it exploits Prolog metaprogramming facilities for reasoning on the protocol itself in order to check whether existential and universal conditions on the protocol hold.

To summarise, West2East integrates the three technologies of agents, WSs and CL for:

- 1 drawing visual AIPs (AUML) and obtaining AIPs’ representations in various formats (Winikoff’s textual notation, XML, Prolog, standard web notations like WS-BPEL and WSDL)
- 2 sharing AIPs through the network (WSs)
- 3 validating AIPs by means of rapid prototyping, and formally verifying them (CL).

Thanks to its features, West2East is aimed at helping different computer science professionals, including software engineers, who want to check properties of an AIP drawn as an AUML interaction diagram. To this end, the software engineer may either use the Prolog-based verification facilities that West2East already offers or adapt one of the outputs of the translation process performed by West2East, to his/her purposes.

3 West2East at a glance

West2East is a computer-aided AOSE tool. It is available to the research community⁶ 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

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 WSSs; 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 by using WS-BPEL and WSDL, like in any standard application based on WSSs, or by using other languages that might be preferred by the agents’ developers in specific domains or situations. Thus, our MASs are composed of 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 is easily met since they are XML-based standard languages) 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 (Gomez-Sanz and Pavon, 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 5. In order to be accessed by the reader agent, the resulting document must be published in some *ad hoc* repository. Besides generating the document representing the AIP, West2East may also be used for automatically generating the code that corresponds to the AIP itself (Section 6).

As for the reader agent, it reads the AIP document published by the provider, in order to ‘acquire’ 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 dialogue with the publisher, as discussed in Section 7.

4 Visual representation of AIPs in AUML

We consider 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 semiformal graphical notation inspired by the UML standard, but not constrained to stick to UML. The general philosophy of AUML, in fact, is:

When it makes sense to reuse portions of UML, then do it; when it doesn't make sense to use UML, use something else or create something new. (The AUML Home Page, <http://www.auml.org/>)

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, which 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.

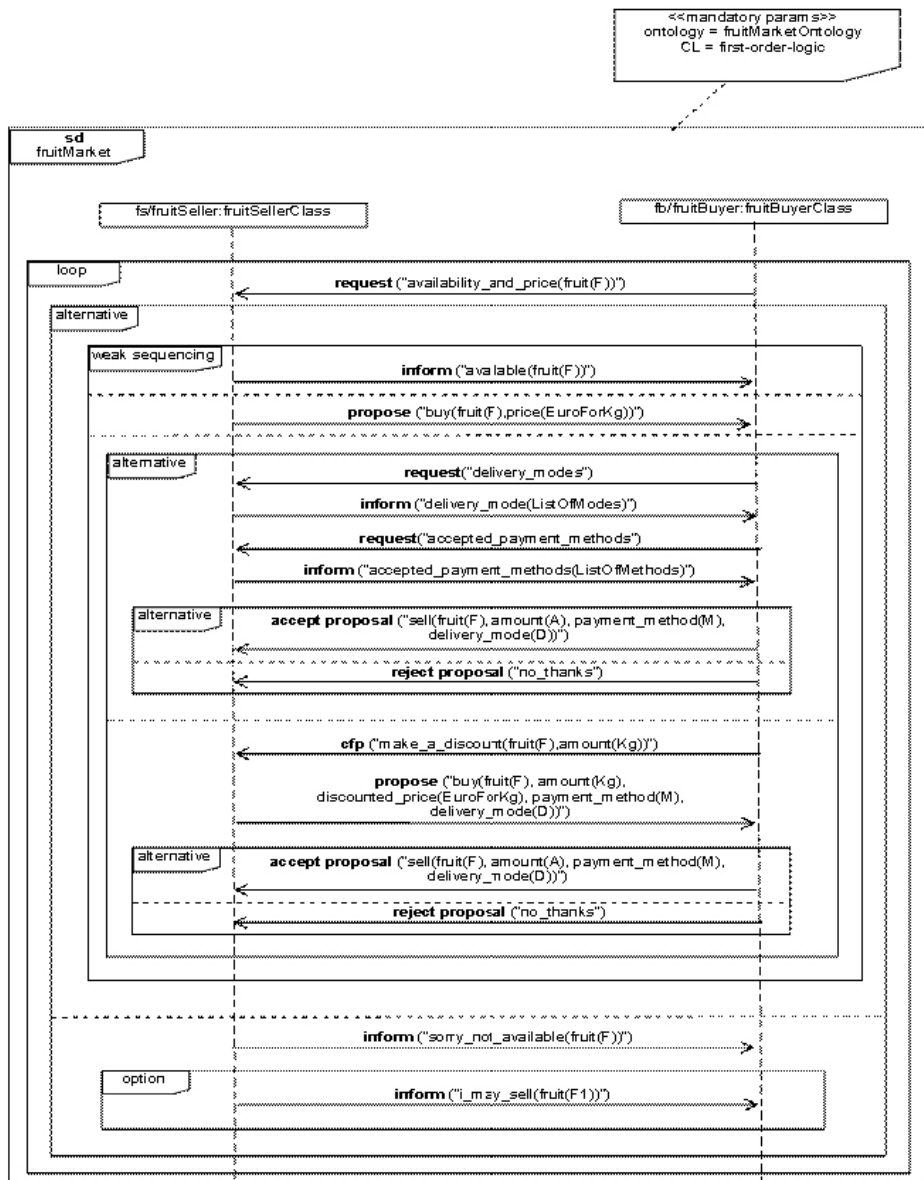
To make an example of an AIP in AUML, Figure 3 describes a `FruitMarket` scenario where a fruit seller (the agent on the left, named `fs`, playing the role of `fruitSeller` and belonging to the `fruitSellerClass`) interacts with a fruit buyer (the agent on the right, named `fb`, playing the role of `fruitBuyer` 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 consists in first informing the buyer that fruit *F* is available, and then proposing the fruit at price *EuroForKg*. At this point, the seller may expect two reactions from the buyer (alternative box):
 - a The buyer prefers to choose 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.

- b The buyer prefers to get a discount, even if this means that it will not be able to choose the delivery and payment methods. Thus, it sends a Call for Proposal (CFP) to the seller, asking to make another proposal with a 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 $F1$ is available instead.

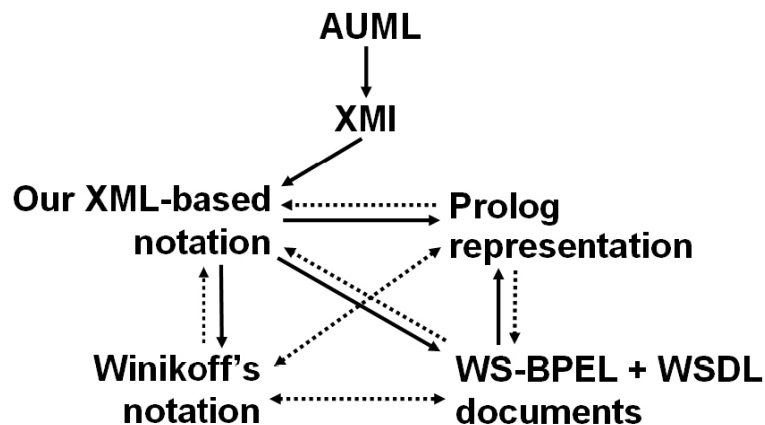
Figure 3 Fruit marketplace protocol in AUML



5 Translation functionalities

Since visual diagrams like AUML AIPs have been conceived and designed to facilitate human beings, but are not suitable for being worked on by machines, we translate them into four different notations amenable to automatic processing. The ‘translation library’ has been developed in Java. It takes in input the XMI representation of the AIP and, by exploiting the Extensible Stylesheet Language Transformations (XSLT) technology (W3C Working Group, 1999), extracts the relevant features of the 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 the translation of the two WS-BPEL and WSDL documents into a Prolog term. This *WS-BPEL + WSDL* \rightarrow *Prolog* translation takes advantage of JDOM.⁷ 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 when the AIP is represented in Winikoff’s notation.

Figure 4 Bidirectional translation between the different notations supported by West2East



5.1 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 that proposed by Koning and Romero-Hernandez (2003). Besides this, it is integrated in the Prometheus Design Tool (PDT),⁸ 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,⁹ although at the moment this functionality does not look at the AIPs. PDT generates code for each agent, capability, plan, event and 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 (*start name*) and the last concludes the protocol (*finish*). Commands in between are used to define agents (*agent shortname longname*), messages (*message*), start and end of boxes (*box and end*), types of the box (*weak sequencing, alternative, option, etc.*), boundaries between regions within a box (*next*), guards (*guard*), continuations (*goto and label*) and the end of an agent’s participation in the protocol (*stop*).

We extend this textual notation with:

- mandatory parameters (*content-language and ontology*)
- optional parameters (to be eventually defined by the developer)
- information on the agent *class* and *role*
- optional content of the messages.

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

```

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))
    message p r propose buy(fruit(F),price (EuroForKg))
  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)
  ...

```

5.2 *Our XML notation*

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

```

<?xml version="1.0" encoding="UTF-8"?>
<protocol-diagram-intermediate-notation
xmlns:UML2="org.omg.xml.namespace.UML2"
xmlns:UML="org.omg.xml.namespace.UML">
<parameters> &lt;parameters&gt;
  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>
    ...
  </combined-fragment>
</main-interaction-fragment>

```

5.3 WSDL and WS-BPEL

We have already anticipated in Section 1 the need for and 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.¹⁰

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 Table 1. AUML is an easily comprehensible visual language that allows one to express the high-level structure of interaction protocols, while WS-BPEL is a quite 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 exchanges, operations requested from the WSs and guard conditions. WS-BPEL is thus powerful enough to represent all the information contained in AUML diagrams.

Table 1 Correspondence between AUML and WS-BPEL concepts

<i>Concept</i>	<i>AUML</i>	<i>WS-BPEL</i>
Roles	ag-name/ag-role: ag-class box	myRole and partnerRole tags
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

To make an example, the AUML protocol depicted in Figure 3 corresponds to the WSDL and WS-BPEL documents partially shown below:

WS-BPEL specification

```

1: <process xmlns="http://schemas.xmlsoap.org/..." ....>
2:   <partnerLinks>
3:     <partnerLink name="publisherPL"
4:       partnerLinkType="lns:SellerBuyer"
5:       myRole="seller" partnerRole="buyer"/>
6:     <partnerLink name="readerPL"
7:       partnerLinkType="lns:BuyerSeller"
8:       myRole="buyer" partnerRole="seller"/>
9:   </partnerLinks>
10:  <variables>
11:    <variable name="continue_1" element="lns:continue_1_type"/>
12:    <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"
18:        portType="lns:publisherPT"
19:        operation="RCV_Mess_1" createInstance="yes" />
20:      <copy><from opaque="yes"/><to>${choose2.value}</to></copy>
21:      <switch>
22:        <case condition="${choose2.value=1}">
23:          ....
24:        n-10: <if condition="${condition_6.value=true}"/><then>
25:          n-9: <invoke partnerLink="publisherPL"
26:            portType="lns:publisherPT"
27:            operation="SND_Mess_15"/>
28:          n-8: <receive partnerLink="readerPL"
29:            portType="lns:readerPT"
30:            operation="RCV_Mess_15"/>

```

```

n-7:      </then> </if>
n-6:      </sequence>
n-5:      </case>
n-4:      </switch>
n-3:      </sequence>
n-2:      <copy><from opaque="yes" />
           <to>${continue1.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 (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 a heterogeneous environment (which a MAS is), it is usually not possible to know in advance which kinds 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 of that protocol branch (`condition = "${continue_1.value} = true"`, for example).

Apart from the first message of the protocol, which, according to the WS-BPEL specification, 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 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 and from the reader's viewpoint. 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*).

5.4 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 functionalities 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 XMC to verify properties of the AIP, using techniques alternative to those described in Section 7. We have just started to experiment with XMC, and we trust that adapting our Prolog notation to it will require a small effort.

To give the flavour of the Prolog notation, a fragment of the FruitMarket AIP is included 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 representations. The only difference that should be noted is the usage of JADE agent identifiers 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 he/she wants to automatically generate the Prolog code of the two agents in such a way that they can be run inside JADE.

```

process(
  parameters(
    ontology('FruitMarketOntology'),
    content_language('first order logic'),
    protocol_name('sd FruitMarket'),
    agent_publisher (short_name('seller@giocas:1099/JADE'),
      long_name('fs/fruitSeller:fruitSellerClass')),
    agent_reader(short_name('buyer@giocas:1099/JADE'),
      long_name('fb/fruitBuyer:fruitBuyerClass')),
    main_fragment (
      while(no_guard,
        seq([
          send(msg('REQUEST','availability_and_price(fruit(F))'),
            switch([
              case(no_guard,
                seq([
                  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)'),
                            ....

```

6 Automatic generation of the protocol-compliant code

Automatic generation of code from specifications allows the MAS engineer to perform a very early testing of the specification without needing to spend time and resources on implementing it. 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 used only to enforce the correct transitions, and transitions correspond either to communicative actions or to the checking of conditions. The generated clauses look like `exec(FromState) :- Transition, exec(ToState)` and model the ability to move from `FromState` to `ToState` by means of `Transition`. States are represented using Prolog terms like `s('sd ProtocolName', 0)` (the state where the protocol starts), `s('sd ProtocolName', final)` (the state where the protocol terminates) and `s(...(s('sd ProtocolName', 0), N), ...)` (intermediate states). Transitions are either communicative acts or conditions to check. They may also be the null transition, thus leading to even simpler clauses, `exec(FromState) :- exec(ToState)`. Despite the simplicity of the generated clauses, some background on Prolog programming might help to understand this and the next sections.¹¹

According to the nature of the AIPs we consider, transitions can be of four kinds: *send*, *receive*, *check* and *null*. 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 italics 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.

```

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

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

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

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

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)).

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

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)).

```

```

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)).

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)).

.....

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('sd FruitMarket',0),0),1),1),1),0)).

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

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)).

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 the Guard atom can be successfully proven in the current agent program (in Prolog words, 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 (`receive`, in clauses 8, 9 and *n-1*) characterises a *send transition* (a *receive transition*, respectively). These communication predicates are implemented by the DCaseLP library (Gungui *et al.*, 2005), which 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 implements the transition. The state that represents the end of the protocol is identified by the constant `final`. The proof of `exec(s(ProtocolId,final))` always succeeds (clause *n*).

6.1 Translating cycles

A `while(Guard,WhileActivities)` action performed in the state `s(S,I)` for reaching the state `SFinal`¹² is translated into:

```

a:  exec(s(S,I)) :- check_guard(s(S,I),Guard),
exec(s(s(S,I),0)).

```

```

b:  exec(s(S,I)) :- exec(SFinal).
c:  exec(s(S,I1)) :- exec(s(S,I)).
    s 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 $s('sd \text{ FruitMarket}', 0)$ for reaching the state $s('sd \text{ FruitMarket}', \text{final})$. Clause 0 of our code fragment corresponds to the first clause of the translation of the while activity (clause a in the general translation schema above); 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 reaching state $s('sd \text{ FruitMarket}', 1)$ (corresponding to $s(S, I1)$ in clause c). When discussing the translation of options, we will see that this truly happens.

6.2 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:

```

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

```

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

6.3 Translating sequences

A $\text{seq}([\text{Activity0}, \dots, \text{ActivityN}])$ action performed in the state $s(S, I)$ for reaching the state $SFinal$ is translated into:

```

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

```

Examples of this translation are clauses 4 and 5. The state $s(S, I)$ from which the translation starts is $s(s('sd \text{ FruitMarket}', 0), 0)$ and the state to reach is $s('sd \text{ FruitMarket}', 1)$. Clause 4 of our running example corresponds to clause a of the general translation schema, while clause 5 corresponds to the translation of the first activity within the sequence, $\text{send}('REQUEST', 'availability_and_price(\text{fruit}(F))', 'seller@giocas:1099/JADE')$, 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.

6.4 Translating alternatives

A `switch([case(Guard0,Activity0), ..., case(GuardN,ActivityN)])` action performed in the state $s(S,I)$ for reaching the state S_{Final} is translated into:

```

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
....
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 a 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.

6.5 Translating options

An `if_then(Guard,Then_activities)` action performed in the state $s(S,I)$ for reaching the state S_{Final} is translated into:

```

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

```

Clauses *n-3* and *n-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 S_{Final} to `s('sd FruitMarket',1)`. *ThenActivities* corresponds 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.

Both the publisher and the reader may take advantage of the automatic code generation functionality. In fact, given the same Prolog AIP (which can be obtained directly from the AUML AIP by the publisher, and from any textual representation of the AIP but Winikoff's 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.

7 Reasoning about the protocol

In order to allow the agents to reason about the protocol, we have implemented a library of Prolog predicates. The reasoning functionality offered by West2East can be exploited by both the publisher and the reader, but it makes sense only for the latter, since the publisher is expected to publish an AIP designed to meet its requirements.

As better explained later in Section 8, when the implemented reader agent obtains the WS-BPEL specification of the AIP required for interacting with the publisher, it first needs to translate it into a representation amenable to being reasoned about. Since we have developed a MAS composed of Prolog agents integrated into the JADE platform, the Prolog reader agent will first need to translate the WS-BPEL specification into Prolog. West2East provides a ‘WS-BPEL to Prolog’ translation routine based on JDOM.

The reader agent may check whether an AIP represented as a Prolog term is correct from a syntactic point of view, by exploiting the `syntax_correctness` (`WS_BPELProtocol`, `SyntaxOutput`) predicate that we have developed. `WS_BPELProtocol` is the representation of the WS-BPEL process into Prolog, and `SyntaxOutput` is a variable that will be unified with either `yes` or `no` after the check has been completed. The syntax check is guided by the structure of the `WS_BPELProtocol` term, whose BNF is given below. The `syntax_correctness` predicate simply checks that `WS_BPELProtocol`’s structure respects this BNF.

```

aip-in-prolog ::= process(parameters(ontology(ontology-name),
    content_language(cl-name)),
    protocol_name(protocol-name),
    agent_publisher(short_name(short-name)),
    long_name(long-name)),
    agent_reader(short_name(short-name),
    long_name(long-name)), main_fragment(structured-activity))

atomic-activity ::= send(FIPA-ACL-message) | receive(FIPA-ACL-message)

structured-activity ::= atomic-activity |
    while(guard, structured-activity) |
    switch(case-activity) |
    switch_otherwise(case-activity, structured-activity) |
    if_then(guard, then-activity) |
    if_then_else(guard, structured-activity, structured-activity) |
    seq(sequence-activity)

sequence-activity ::= [] |
    [structured-activity |prolog sequence-activity]

case-activity ::= [] |
    [case(guard, structured-activity) |prolog case-activity]

FIPA-ACL-message ::= msg(FIPA-ACL-performative, content)

FIPA-ACL-performative ::= "ACCEPT-PROPOSAL" | "AGREE" |
    "CANCEL" | ...

guard ::= no_guard | atom

```

Entities ending with *name* are strings, namely sequences of characters enclosed in " or ', as is *content*. Terminal symbols are in bold. ' $|_{prolog}$ ' is the Prolog list constructor, usually denoted by '[' (we used a different notation to avoid confusion with the '[' symbol used in the right-side BNF rules). *Atom* is a Prolog atom.

If the syntax correctness check terminates successfully by unifying `SyntaxOutput` with `yes`, the reader agent may generate the Prolog program starting from the Prolog representation of the AIP, which we have indicated with the `WS_BPELProtocol` logical variable. It may thus call the `generate_code(Reader, WS_BPELProtocol, PrologCode, ProtocolId)` predicate, where `PrologCode` is a free logical variable that will be unified with the list of clauses generated following the algorithm described in Section 6, and `ProtocolId` is the identifier of the protocol ('sd FruitMarket' in our running example; it must be instantiated when the predicate for code generation is called).

In order to reason about the AIP, the reader agent must first assert the clauses belonging to the Prolog program (`PrologCode`) obtained by calling the `generate_code` predicate. Note that *asserting* these clauses into the agent's working memory does not mean *executing* the programme. However, asserting them is necessary in order to perform the reasoning stage. We have developed an `assert_protocol_clauses(PrologCode)` predicate that simply calls the `assert` built-in predicate for each clause in the `PrologCode` list.

The actual reasoning stage is performed by the `check_conditions(ProtocolId, Conditions, CondOutput, Traces)` predicate. The developer of the agent must instantiate the `Conditions` variable with a list containing `exists(Action)` and/or `forall(Action)` terms. If the list is empty, no check on the protocol is made. `CondOutput` will be unified with either `yes` or `no`, according to the reasoning outcome, and `Traces` will be unified with a list of execution traces, one for each condition to be verified in the `Conditions` list. For each condition, the corresponding trace may be either one trace where the condition is not satisfied (as a negative witness for a `forall(Action)` test) or one trace where the condition is satisfied (as a positive witness for an `exists(Action)` test). `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 (in any possible path) a `send(Performative, Content, Receiver)`, or a `receive(Performative, Content, Sender)`, or a `check_guard(State, Guard)` is executed. In Prolog technical terms, the reasoning stage is implemented by allowing the reader agent to exploit Prolog metaprogramming capabilities by making a depth-first exploration of the SLD-tree for $P \cup \{G\}$ via R , where P is the program generated by the `generate_code` predicate and asserted into memory with the `assert_protocol_clauses` predicate, G is the goal that starts the execution of the protocol (`exec(s('sd FruitMarket', 0))` in our example) and R is the left-most 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).

The implementation of `check_conditions(ProtocolId, Conditions, CondOutput, Traces)` calls the predicate for checking just one condition, `check_one_cond(ProtocolId, Condition, Output, Trace)`, on each condition belonging to the `Conditions` list, until either the proof for one of them fails or there are no more conditions to test.

`check_one_cond(ProtocolId, Condition, Output, Trace)` exploits the auxiliary predicate `forall_solve(exec(s(ProtocolId,0)), C, BadTrace)` if the condition to check is of type `forall(C)`, and exploits `exists_solve(exec(s(ProtocolId,0)), C, GoodTrace)` if the condition to check is of type `exists(C)`.

`forall_solve` succeeds if there is one execution trace where the final state is reached before the condition to test is satisfied; this trace is unified with the third argument, as shown in the predicate implementation given below. The `move(Node, Statement1, Node1)` predicate succeeds if the agent's program includes an `'exec(Node) :- Statement1, exec(Node1)'` or an `'exec(Node) :- exec(Node1)'` clause (in the last case, `Statement` is unified with `nil`).

```
forall_solve(Node, Cond, Solution) :-
forall([], (nil, Node), Cond, Solution).

forall(Path, (Statement, Node), Cond, [(Statement, Node) |Path]) :-
forall_final((Statement, Node), Cond).

forall(Path, (Statement, Node), Cond, Solution) :-
move(Node, Statement1, Node1),
Statement1 \= Cond,
Node1 \= Cond,
not(member((Statement1, Node1), Path)),
forall([(Statement, Node) |Path], (Statement1, Node1), Cond,
Solution).

forall_final((Statement, exec(s(Something, final))), Cond) :-
Cond \= Statement,
Cond \= exec(s(Something, final)) .
```

Thus, if `forall_solve` succeeds, the condition is not satisfied, and `check_one_cond` must unify the `Output` variable with `no`, and the execution trace with the one where the condition was not satisfied (first clause below). Otherwise, if `forall_solve` fails, the second clause below defining `check_one_cond` is used and the `Output` variable is unified with `yes`. The execution trace is unified with the empty list, since there are no negative witnesses.

```
check_one_cond(ProtocolId, forall(C), no, (forall(C), BadTrace)):-
forall_solve(exec(s(ProtocolId,0)), C, BadTrace), !.

check_one_cond(ProtocolId, forall(C), yes, (forall(C), [])).
```

On the other hand, `exists_solve` succeeds if one trace where the condition is verified is found. Its implementation is given as follows:

```
exists_solve(Node, Cond, Solution) :-
exists([], (nil, Node), Cond, Solution).
```

```

exists(Path, (Statement, Node), Cond, [(Statement, Node)|Path]) :-
exists_final((Statement, Node), Cond).

exists(Path, (Statement, Node), Cond, Solution) :-
move(Node, Statement1, Node1),
not(member((Statement1, Node1), Path)),
exists([(Statement, Node)|Path], (Statement1, Node1), Cond,
Solution).

exists_final(_, Cond), Cond).

exists_final((Cond, _), Cond).

```

If `exists_solve` succeeds, it means that there is at least one trace where the condition is met. Thus, `check_one_cond` must unify the `Output` variable with `yes`, and the `Trace` variable with the successful trace found by `exists_solve` (first clause below). Otherwise, if no trace where the condition is met can be found, the `Output` variable is unified with `no`, and the `Trace` variable with the empty list (second clause).

```

check_one_cond(ProtocolId, exists(C), yes, (exists(C), GoodTrace)) :-
exists_solve(exec(s(ProtocolId, 0)), C, GoodTrace), !.

check_one_cond(ProtocolId, exists(C), no, (exists(C), [])).

```

This reasoning stage is performed offline, before the reader agent starts to interact with the publisher one. Since the reasoning stage gives a set of traces (bad traces witnessing when universal conditions are not verified, and good traces witnessing when existential conditions are verified) as its output, the reader agent may decide what to do according to the outcome of this offline reasoning stage, and according to the trace associated with each condition to check. In particular, it may select one of the obtained good traces, and try to enforce the protocol execution to follow that trace. The `start_protocol` predicate, which starts the actual interaction with the publisher agent, takes the chosen trace as its second argument (the first one is the protocol identifier). The executing agent will perform those actions that allow it to follow the chosen trace, as long as the choice belongs to it. For example, if at a certain point of the protocol execution (or, in other words, ‘in a certain state’), the reader agent may send either message M_1 or message M_2 , it will choose to send the message foreseen by the execution trace to enforce, namely the message that, according to the previous reasoning activity, will allow the agent to interact in such a way as to satisfy the condition to meet. However, the execution trace to enforce contains activities, such as receiving a given message M_3 , whose fulfilment is outside the capabilities of the agent (which will receive the message sent by the partner agent, not the message that it hopes to receive). In case of reception of a message allowed by the protocol, but different from the expected one, according to the chosen execution trace to enforce, the reader agent just gives up enforcing the trace, and continues its interaction following a different protocol branch.

The reader agent is implemented by the MAS developer, which may take advantage of the automatic code generation and reasoning functionalities offered by `West2East` and described above. If the developer wants to take advantage of them, he/she should arrange them into the reader’s code in the following way (`select_trace` must be defined by the developer):

```

syntax_correctness(WS_BPELProtocol, yes),
generate_code(reader, WS_BPELProtocol, PrologCode, ProtocolId),
assert_protocol_clauses(PrologCode),
check_conditions(ProtocolId, Conditions, yes, Traces),
select_trace(Traces, Trace),
start_protocol(ProtocolId, Trace)

```

To go on with our fruit market example, let us consider a ‘restrictive’ fruit buyer agent that accepts interaction with the fruit seller agent only if it provides – whatever the protocol branch followed – a bunch of payment methods among which the buyer can choose. The fruit buyer agent’s code should contain the fact: `check_conditions('sd FruitMarket', [forall(receive('INFORM', 'accepted_payment_methods(ListOfMethods)', Sender))], Out, Traces)`. This universal condition is not verified by the protocol depicted in Figure 3, since there are two branches where the reader agent, namely the fruit buyer, will not receive the `'accepted_payment_methods(ListOfMethods)'` message: the branch where the buyer sends a call for proposal message, `cfp(make_a_discount(fruit(F), amount(Kg)))`, and the branch where the seller has no fruit to sell and sends an `inform(sorry_not_available(fruit(F)))` message. Thus, `Out` would be unified with `no`, and `Traces` with the empty list. 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 from more than one payment method. It would then check the existential condition `exists(receive('INFORM', 'accepted_payment_methods(ListOfMethods)', Sender))`. The protocol will verify this condition, and the buyer agent would start the protocol execution, trying to follow the execution trace that leads to the verification of the condition.

While executing, the buyer explains its actions by printing them on a log file. The following fragment describes the intended and the actual behaviour of the ‘flexible’ fruit buyer.

```

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

check_guard(s('sd FruitMarket',0), noguard)
send('REQUEST', 'availability_and_price(fruit(F))',
'seller@giocas:1099/JADE')
check_guard(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('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')

```

```

I executed the statement check_guard(s('sd FruitMarket',0),no_guard)
.....
Nondeterministic action: I received the message I was waiting for,
('INFORM','available(fruit(F))','seller@giocas:109 9/JADE')
.....
Nondeterministic action: I received the message I was waiting for,
('INFORM','delivery_mode(ListOfModes)','seller@giocas:1099/JADE')

I executed the statement
send('REQUEST',accepted_payment_methods,
'seller@giocas:1099/JADE')

Nondeterministic action: I received the message I was waiting for,
('INFORM','accepted_payment_methods(ListOfMethods)',
'seller@giocas:1099/JADE')

I have reached my goal!

```

8 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.

In 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 on both WSs and JADE (and also JADE agent systems that exploit languages for WSs, as in this example), it is independent from any specific infrastructure.

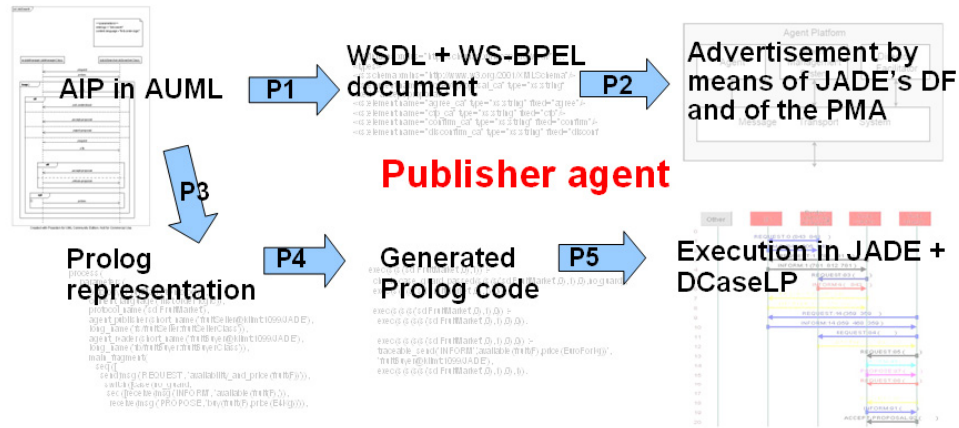
To implement our publisher/reader MAS, we have developed a JADE Protocol Manager Agent (PMA) that, together with JADE’s Directory Facilitator (DF), allows agents to publish and retrieve WSDL + WS-BPEL specifications representing AIPs.

The development of the publisher agent must take the following activities into account (see Figure 5; the ‘P’ before the activity number stands for ‘publisher’):

- P0 design of an AIP in AUML following an AOSE methodology (outside the scope of West2East)
- P1 translation from the AUML visual diagram to the WSDL + WS-BPEL notation discussed in Section 5.3
- P2 advertisement of the WSDL + WS-BPEL document by exploiting the services offered by the DF and the PMA, as described later in this section
- P3 translation from the AUML visual diagram into the Prolog notation discussed in Section 5.4

- P4 generation of the Prolog code from the Prolog term representing the AIP, as described in Section 6
- P5 execution of the Prolog code in JADE extended with the DCaseLP libraries.

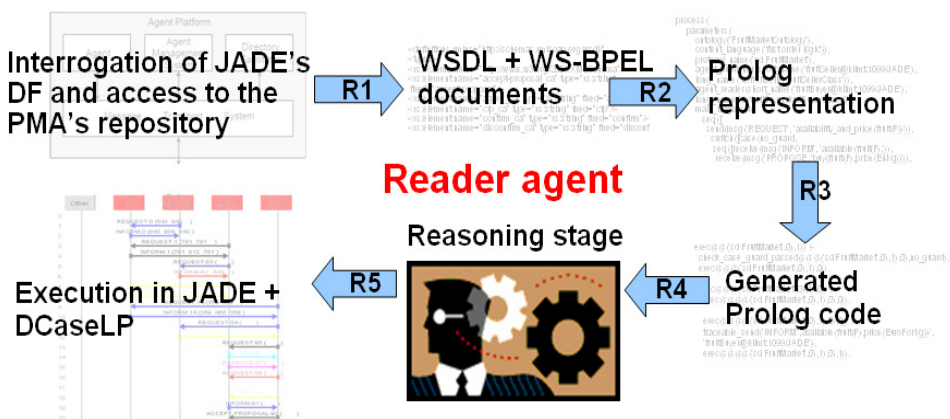
Figure 5 Activities for the development of a publisher agent in JADE using West2East



The reader agent must be engineered in order to follow the steps shown in Figure 6:

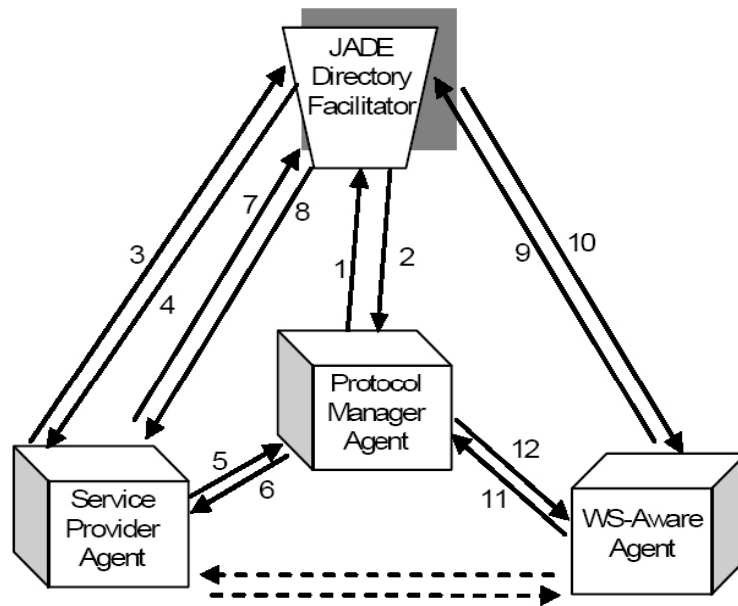
- Step R1 interrogation of JADE's DF and access to the repository managed by the PMA in order to retrieve the WSDL + WS-BPEL documents
- Step R2 translation from the WSDL + WS-BPEL notation to the Prolog one
- Step R3 generation of the Prolog code from the Prolog term
- Step R4 reasoning stage discussed in Section 7
- Step R5 execution of the Prolog code in JADE extended with the DCaseLP libraries.

Figure 6 Activities for the development of a reader agent in JADE using West2East



For supporting the interaction between the publisher and the reader, we have designed and implemented the MAS depicted in Figure 7. We will name as ‘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 5.3.

Figure 7 Our MAS implemented in JADE



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 MySQLDBMS.¹³

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 7). When a publisher agent wants to advertise an agent service offered by means of an AIP initially specified in AUML, it:

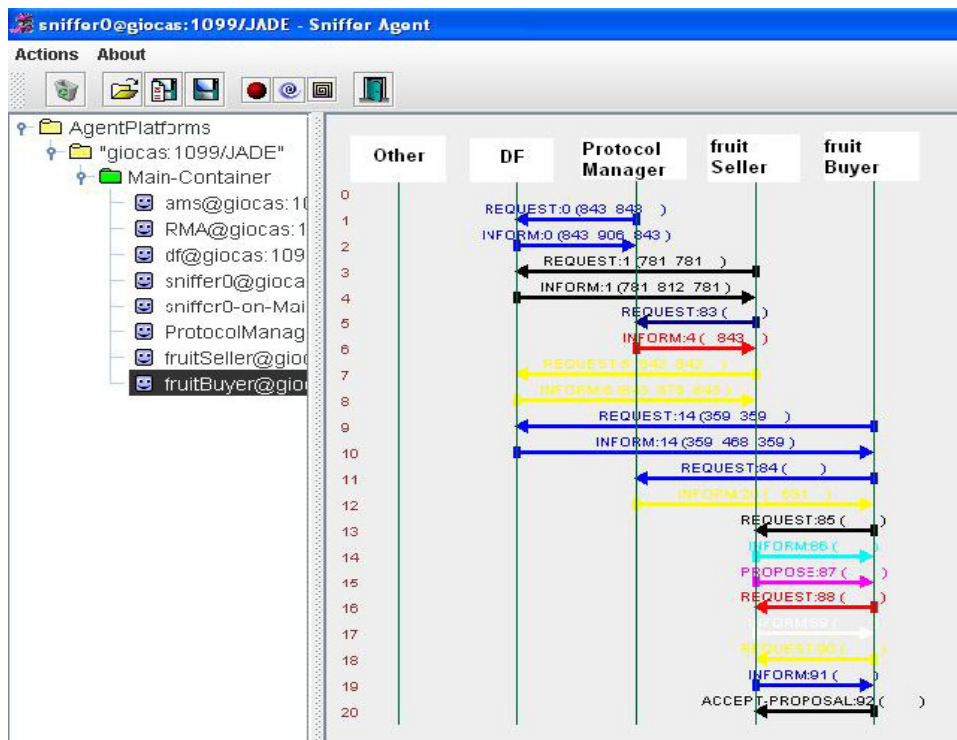
- generates the WS-BPEL + WSDL documents from the AUML AIP
- looks in the DF to find the protocol-publishing agent service (arrows 3, 4)
- sends a message to the PMA with the WSDL + WS-BPEL document, and waits to receive the Protocol Identifier (PID) assigned by the PMA to it (arrows 5, 6)
- registers 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:

- 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)
- 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
- from the WSDL + WS-BPEL specification, generates the corresponding Prolog term, and, from this, the Prolog code, and reasons about it
- eventually starts, according to the reasoning outcome, 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 7).

Figure 8 refers to an execution run of the MAS composed of 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 12 messages correspond to the communication represented by the 12 solid arrows in Figure 7, 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.

Figure 8 Execution run in JADE



9 Related work

The related work that we consider in this section deals with the three features that best characterise West2East: the exploitation of AUML for engineering WSs, the automatic generation of code and textual documents starting from the AUML specification, and the capability of reasoning about AIPs and WSs. The output of our analysis of the related work may be summarised in the following way:

- There are no implemented tools that, starting from AUML sequence diagrams, generate a code compliant with standards for WSs (as, on the other hand, West2East does).
- There are no implemented tools that, starting from AUML sequence diagrams, generate a corresponding specification in Winikoff's notation (as, on the other hand, West2East does).
- There is only one implemented tool (Ingenias Development Kit) that, starting from AUML sequence diagrams, generates an executable Prolog code (as West2East does).
- There are many implemented tools suitable (or easily adaptable) for reasoning about interaction protocols, but none can be fed with the specification of an AIP given in as many languages as those supported by West2East.

9.1 Exploiting AUML for engineering WSs

Although the literature describes various approaches either for modelling WSs using UML or for generating documents in BPEL (be it WS-BPEL, or its previous version, BPEL4WS) starting from UML diagrams, to the best of our knowledge no proposals exist for generating WSDL and WS-BPEL documents starting from an AUML AIP specification. In this section, we review the state-of-the-art usage of UML for engineering WSs. However, our approach goes beyond the current state of the art since our starting point is not UML, but its agent-oriented extension, AUML. Also, while many 'UML → BPEL' proposals are not supported by any implemented tool, West2East offers a set of libraries that implement the 'AUML → BPEL' translation. These libraries are available to the research community and demonstrate the feasibility of our approach.

Grønmo and Solheim (2004) and Bauer and Huget (2004) both exploit UML for modelling WS composition, and propose extensions to UML when it does not provide enough support for the given purpose. The extensions suggested by Grønmo and Solheim refer to 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. Both sets of authors, Grønmo and Solheim, and Bauer and Huget, consider business processes specified using UML activity diagrams, while we start from AUML interaction diagrams. We share with Bauer and Huget the target languages into which UML activity diagrams should be translated, WSDL and BPEL, while the target of Grønmo and Solheim is a workflow XML language. Provost (2003) proposes a general approach to designing WSs in UML, expresses their semantics in WSDL, and implements them in a (not specified) supporting programming language.

Provost's main goal is to make the design and implementation of WSs more rational and simple, through the use of UML class and sequence diagrams. No implemented tool for generating the WSs specification from the UML one is provided by the above authors.

Gardner (2003) describes a UML profile which allows business 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 diagrams and generates the corresponding BPEL4WS and WSDL files. Gardner uses UML class and activity diagrams, suitable for representing in a concise way WS interactions in business process, while we use UML interaction diagrams, suitable for representing messages exchanged by agents in detail. Mantell (2005) also concentrates his efforts on UML activity diagrams and describes a tool which takes processes defined by these diagrams and generates the corresponding BPEL4WS and WSDL files. The key difference between our work and both Gardner's and Mantell's is that we translate interaction protocols represented as AUML protocol diagrams 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,¹⁴ no longer available.

Besides these proposals, there are many commercial toolkits that support the development of WSs by providing editors for drawing visual representations of the business processes and for automatically generating the WSDL and BPEL code from them. They usually consist of a development toolkit and a BPEL engine. The BPEL Maestro Toolkit,¹⁵ Websphere Studio¹⁶ and Oracle BPEL Process Manager (BPEL PM)¹⁷ are just some of them. Among them, only BPEL Maestro allows the developer to represent business processes using UML. However, it supports no agent-oriented extensions of UML.

Table 2 Exploitation of AUML for engineering WSs: comparing West2East to other approaches

<i>Author</i>	<i>Tool</i>	<i>Input</i>	<i>Output</i>	<i>WS</i>	<i>AG</i>
Casella and Mascardi (2007)	West2East	AUML sequence diagram	WSDL and WS-BPEL; Prolog; XML; Winikoff's representation	Yes	Yes
Grønmo and Solheim (2004)	None	UML activity diagram	Workflow XML language	Yes	No
Bauer and Huget (2004)	None	UML activity diagram	WSDL and BPEL4WS	Yes	Yes
Provost (2003)	None	UML class and sequence diagrams	WSDL; an unspecified programming language	Yes	No
Gardner (2003) and Mantell (2005)	A plug-in integrated in the IBM Emerging Technologies Toolkit	UML class and activity diagrams	BPEL4WS	Yes	No
Parasoft	BPEL Maestro Toolkit	UML activity diagram	BPEL	Yes	No

Table 2 provides a summary of the comparison between West2East and other existing proposals along the following dimensions: *Tool*, the implemented toolkit, if any, that supports the engineering approach; *Input*, the visual diagram taken as input by the

proposed approach, in order to specify the business process; *Output*, the output language for representing the visual diagram in a standard notation for WSS; *WS*, 'is the approach oriented towards WSS?'; *AG*: 'Does the approach take agent-oriented extensions of UML into account?'

From this comparison, it turns out that Bauer and Huget's proposal is the one closest to ours with respect to the above criteria; however, they do not provide any implemented toolkit supporting their approach.

9.2 Generating executable code and textual documents from AUML

The generation of executable code and of documents in textual notation, amenable for machine processing, from an AUML specification has been dealt with by various papers and implemented toolkits. Dinkloh and Nimis (2003) provide an implemented tool for translating an AUML sequence diagram into a finite state machine, while Huget (2002) proposes a semiautomatic approach for generating the Java code to implement agents whose interaction is described, again, by an AUML sequence diagram. The INGENIAS Development Kit, IDK,¹⁸ is a tool supporting the INGENIAS MAS design methodology defined by Gomez-Sanz and Pavon (2003). The IDK includes the INGENIAS Editor for defining specifications of a MAS using generic agent concepts and diagrams, including AUML-like protocol diagrams. It also includes several code generation modules, among them the JADE protocol generator, which generates JADE agents that implement protocols defined with INGENIAS diagrams, and the Prolog generator, a basic, noncomplete translation of INGENIAS elements to prolog predicates. The PASSI Toolkit, PTK (Chella *et al.*, 2003), shares a similar aim with IDK. It supports the design of a JADE system using the PASSI methodology (Cossentino and Potts, 2002) that refers to the most common standards in software engineering and agents (UML, AUML, FIPA, XML, RDF). Romero-Hernandez and Koning (2004) describe a graphic editor that allows the definition of arbitrarily shaped graphic artefacts, including AUML sequence diagrams, using an XML script, and the definition of how the artefacts are going to be translated to textual representations suitable to be further refined into the Promela language and input to the SPIN model checker. Also, VIPER (Rooney *et al.*, 2004) provides a suite of graphical tools that allows users to develop AIPs using the AUML conventions. When the AIP has been designed through a process of user-driven code generation, VIPER generates a code for the Agent Factory Agent Programming Language (AF-APL), a language that sits at the heart of the Agent Factory (AF) framework (Collier *et al.*, 2003). Ehrler and Cranefield (2004) developed PAUL, a Plug-in for AUML Linking which allows agent developers to control agent conversation by linking application-specific code to AUML sequence diagrams, which are then interpreted automatically at runtime. PAUL is built using the FIPA-compliant Java-based agent platform Opal (Purvis *et al.*, 2002). Finally, the approach proposed by Cabac and Moldt (2004) is aimed at translating AUML diagrams into Petri Nets. The authors provide tool support for their approach by combining a tool for net components with a tool for drawing AIP diagrams. This combined tool is available as a plug-in for Renew (Reference Net Workshop) (Kummer *et al.*, 2004).

Table 3 summarises the main features of the above proposals, following the criteria already introduced for Table 2. For the sake of clarity, we include West2East also in Table 3. None of the proposals considered in Table 3, nor those considered in Table 2, produces documents in a standard language for WSS starting from AUML diagrams.

Also, there is no tool that generates a document in Winikoff's representation starting from the corresponding AUML diagram. This feature, provided by West2East, might be exploited for generating the JACK code from AUML sequence diagrams, using PDT and Winikoff's notation as a bridge. A piece of Prolog code is output, in a noncomplete way, only by the IDK. The Prolog code generated by West2East, on the other hand, is complete, apart from those details that are missing in the specification itself (namely, the initial state of the agent and the mechanisms for checking conditions). Finally, the XML format that West2East outputs is mainly used for the internal purposes of our environment, as an intermediate format between the different notations that we provide. However, it may also be used as it is, since our XML notation is easy to read and understand both for a human being and for a machine. Since none of the toolkits considered in Table 3 outputs an XML representation of the input AUML diagram, this feature is also an original contribution of West2East.

Table 3 Generation of executable code and textual documents from AUML: comparing West2East to other approaches

<i>Author</i>	<i>Tool</i>	<i>Input</i>	<i>Output</i>	<i>WS</i>	<i>AG</i>
Casella and Mascardi (2007)	West2East	AUML sequence diagram	WSDL and WS-BPEL; Prolog; XML; Winikoff's Representation	Yes	Yes
Dinkloh and Nimis (2003)	Eclipse plug-in	AUML sequence diagram	Finite State Machine; JADE	No	Yes
Huget (2002)	None	AUML sequence diagram	Java	No	Yes
Rooney <i>et al.</i> (2004)	VIPER	AUML sequence diagram	Agent Factory Agent Programming Language	No	Yes
Cabac and Moldt (2004)	Renew plug-in	AUML sequence diagram	Petri Net	No	Yes
Gomez-Sanz and Pavon (2003)	INGENIAS Development Kit, IDK	AUML sequence diagram and other diagrams for MAS	JADE; Prolog; HTML	No	Yes
Chella <i>et al.</i> (2003)	Passi Toolkit, PTK	AUML sequence diagram and other UML diagrams	JADE	No	Yes
Romero-Hernandez and Koning (2004)	Unnamed	AUML sequence diagram	Customisable Textual Representation	No	Yes
Ehrler and Cranefield (2004)	PAUL (Plug-in for Agent UML Linking)	AUML sequence diagrams	Java	No	Yes

9.3 Reasoning about AIPs and WSs

West2East Prolog-related functionalities have the same purpose of the implementation of a logic-based protocol language in the theorem prover Isabelle, discussed by Brak *et al.* (2004).¹⁹ The properties of the protocol that can be proven inside West2East are existential and universal ones, whereas those considered by Brak *et al.* include termination and consistency properties. Although limited to checking existential and universal properties, West2East may be exploited for reasoning about AIPs in general,

and for reasoning about WSs in particular. The need for reasoning about WSs is extremely pressing, since the full exploitation of the WS technology requires us to address issues of security and privacy protection, which the West2East reasoning mechanism helps to address. 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. Examples of these proposals include that by Baldoni *et al.* (2005), who describe 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 proven to conform to the protocol. Brogi *et al.* (2004) formalise the WSCI language (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 interoperate 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), who define a notion of conformance between choreography and orchestration which allows them 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.

10 Future work and conclusions

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

- First, any AIP involves 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 *P1* may play the publisher role in AIP *P2*, 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 MAS 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 (or provider respectively) following different interaction protocols. However, an agent can neither be both a provider and a consumer within the same AIP nor a consumer (or provider) for more than one provider (or consumer respectively) within the same AIP. Also, interleaved execution of two or more AIPs is not allowed. We are currently working to overcome 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 ones, but not from documents in Winikoff's textual notation. The implementation of a bidirectional translation facility from any supported notation to any other, which 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, provided that the MAS developer edits it, manually inserts a belief base (namely, a set of Prolog atoms) representing the initial mental state of the agent, and manually adds 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, which is performed by exploiting metaprogramming facilities offered by Prolog and which requires us to express the conditions upon which to perform the reasoning, as Prolog terms. We are working both to make the underlying Prolog code as transparent to the MAS developer as possible, by means of an intuitive graphical interface, and to generate code in languages other than Prolog, first of all Java.

The three main directions of our future work involve overcoming the above limitations, extending the set of notations and languages that West2East supports, both for representing AIPs and for implementing agents, and letting ‘real’ users evaluate West2East on some future application larger than those developed by us. For 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 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 saved and reduction in error gained for people unskilled in AOSE (which both students and practitioners from the industry are) by using West2East in conjunction with a principled AOSE approach. After this stage and the consequent improvements that we will make to West2East, we will start to promote its usage inside those companies with which we actively collaborate.

To conclude, 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 supports the MAS implementation starting from previous analysis and design stages that lead 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.

Acknowledgements

This work was partially supported by the Italian project MIUR PRIN 2005 ‘Specification and verification of agent interaction protocols’. The authors thank the anonymous reviewers for their constructive comments.

References

- Agentcities Task Force (2003) *Integrating Web Services into Agentcities Recommendation*.
- Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Montali, M., Storari, S. and Torroni, P. (2006) 'Computational logic for run-time verification of web services choreographies: exploiting the SOCS-SI tool', *Third International Workshop on Web Services and Formal Methods, WS-FM 2006, Proceedings, Lecture Notes in Computer Science*, Springer, pp.58–72.
- Alberti, M., Daolio, D., Torroni, P., Gavanelli, M., Lamma, E. and Mello, P. (2004) 'Specification and verification of agent interaction protocols in a logic-based system', in H. Haddad, A. Omicini, R.L. Wainwright and L.M. Liebrock (Eds.) *ACM Symposium on Applied Computing (SAC 2004), Proceedings*, ACM, pp.72–78.
- Baldoni, M., Baroglio, C., Martelli, A. and Patti, V. (2003) 'Reasoning about conversation protocols in a logic-based agent language', in A. Cappelli and F. Turini (Eds.) *Advances in Artificial Intelligence, 8th Congress of the Italian Association for Artificial Intelligence, AI*IA 2003, Proceedings, Volume 2829 of Lecture Notes in Computer Science*, Springer, pp.300–311.
- Baldoni, M., Baroglio, C., Martelli, A. and Patti, V. (2006) '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*, January 2007, Vol. 70, No. 1, pp.53–73.
- Baldoni, M., Baroglio, C., Martelli, A., Patti, V. and Schifanella, C. (2005) 'Verifying the conformance of web services to global interaction protocols: a first step', in M. Bravetti, L. Kloul and G. Zavattaro (Eds.) *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW2005, and International Workshop on Web Services and Formal Methods, WS-FM 2005, Proceedings, Volume 3670 of Lecture Notes in Computer Science*, Springer, pp.257–271.
- Bauer, B. and Huget, M-P. (2004) 'Modelling web service composition with UML 2.0', *Int. J. Web Eng. Technol.*, Vol. 1, No. 4, pp.484–501.
- Bauer, B., Muller, J.P. and Odell, J. (2000) 'Agent UML: a formalism for specifying multiagent software systems', in P. Ciancarini and M. Wooldridge (Eds.) *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Revised Papers, Volume 1957 of Lecture Notes in Computer Science*, Springer, pp.91–104.
- Bellifemine, F., Poggi, A. and Rimassa, G. (2000) 'Developing multi-agent systems with JADE', in C. Castelfranchi and Y. Lesperance (Eds.) *Intelligent Agents VII. Agent Theories Architectures and Languages, Seventh International Workshop, Proceedings, Volume 1986 of Lecture Notes in Computer Science*, Springer, pp.89–103.
- Bozzano, M. and Delzanno, G. (2002) 'Automated protocol verification in linear logic', *Fourth International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2002, Proceedings*, ACM, pp.38–49.
- Bozzano, M. and Delzanno, G. (2004) 'Automatic verification of secrecy properties for linear logic specifications of cryptographic protocols', *J. Symb. Comput.*, Vol. 38, No. 5, pp.1375–1415.
- Bozzo, L., Mascardi, V., Ancona, D. and Busetta, P. (2005) 'CooWS: adaptive BDI agents meet service-oriented programming', in P. Isaias and M.B. Nunes (Eds.) *IADIS International Conference WWW/Internet 2005, ICWI 2005, Proceedings*, IADIS Press, Vol. 2, pp.205–209.
- Brak, R.L., Fleuriot, J.D. and McGinnis, J. (2004) 'Theorem proving for protocol languages', *Second European Workshop on Multi-Agent Systems, EUMAS 2004, Proceedings*.
- Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F. and Mylopoulos, J. (2004) 'Tropos: an agent-oriented software development methodology', *Autonomous Agents and Multi-Agent Systems*, Vol. 8, No. 3, pp.203–236.
- Brogi, A., Canal, C., Pimentel, E. and Vallecillo, A. (2004) 'Formalizing web service choreographies', *Electr. Notes Theor. Comput. Sci.*, Vol. 105, pp.73–94.

- Buhler, P.A. and Vidal, J.M. (2005) 'Towards adaptive workflow enactment using multi-agent systems', *Information Technology and Management*, Vol. 6, pp.61–87.
- Bultan, T., Su, J. and Fu, X. (2006) 'Analyzing conversations of web services', *IEEE Internet Computing*, Vol. 10, No. 1, pp.18–25.
- Busi, N., Gorrieri, R., Guidi, C., Lucchi, R. and Zavattaro, G. (2005) 'Choreography and orchestration: a synergic approach for system design', in B. Benatallah, F. Casati and P. Traverso (Eds.) *Service-Oriented Computing, Third International Conference, ICSOC 2005, Proceedings, Volume 3826 of Lecture Notes in Computer Science*, Springer, pp.228–240.
- Cabac, L. and Moldt, D. (2004) 'Formal semantics for AUML agent interaction protocol diagrams', in J. Odell, P. Giorgini and J.P. Muller (Eds.) *Agent-Oriented Software Engineering Workshop, Fifth International Workshop, AOSE 2004, Revised Selected Papers, Volume 3382 of Lecture Notes in Computer Science*, Springer, p.4761.
- Cernuzzi, L. and Zambonelli, F. (2004) 'Experiencing AUML in the Gaia methodology', *Sixth International Conference on Enterprise Information Systems, ICEIS 2004, Proceedings*, Vol. 3, pp.283–288.
- Cervenka, R., Trencansky, I. and Calisti, M. (2005) 'Modeling social aspects of multi-agent systems: the AML approach', *Agent-Oriented Software Engineering Workshop, Sixth International Workshop, AOSE 2005, Proceedings*.
- Chella, A., Cossentino, M. and Sabatucci, L. (2003) 'Designing JADE systems with the support of case tools and patterns', *Exp Journal*, Vol. 3, No. 3, pp.86–95.
- Ciancarini, P. and Wooldridge, M. (2000) 'Agent-oriented software engineering: the state of the art', in P. Ciancarini and M. Wooldridge (Eds.) *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000, Revised Papers, Volume 1957 of Lecture Notes in Computer Science*, Springer, pp.1–28.
- Collier, R.W., O'Hare, G.M.P., Lowen, T.D. and Rooney, C. (2003) 'Beyond prototyping in the factory of agents', in V. Marik, J.P. Muller and M. Pechoucek (Eds.) *Multi-Agent Systems and Applications III, Third International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Proceedings, Volume 2691 of Lecture Notes in Computer Science*, Springer, pp.383–393.
- Cossentino, M. and Potts, C. (2002) 'A CASE tool supported methodology for the design of multi-agent systems', *Software Engineering Research and Practice, International Conference, SERP'02, Proceedings*.
- Cui, B., Dong, Y., Du, X., Kumar, K.N., Ramakrishnan, C.R., Ramakrishnan, I.V., Roychoudhury, A., Smolka, S.A. and Warren, D.S. (1998) 'Logic programming and model checking', in C. Palamidessi, H. Glaser and K. Meinke (Eds.) *Principles of Declarative Programming, Tenth International Symposium, PLILP'98, Proceedings, Volume 1490 of Lecture Notes in Computer Science*, Springer, pp.1–20.
- Denti, E., Omicini, A. and Ricci, A. (2005) 'Multi-paradigm Java-Prolog integration in tuProlog', *Sci. Comput. Program.*, Vol. 57, No. 2, pp.217–250.
- Dinkloh, M. and Nimis, J. (2003) 'A tool for integrated design and implementation of conversations in multiagent systems', in M. Dastani, J. Dix and A.E. Fallah-Seghrouchni (Eds.) *Programming Multi-Agent Systems, First International Workshop, PROMAS 2003, Selected Revised and Invited Papers, Volume 3067 of Lecture Notes in Computer Science*, Springer, pp.187–200.
- Dix, J., Sadri, F. and Satoh, K. (Eds.) (2003) 'Special issue on computational logic in multi-agent systems', *Annals of Mathematics and Artificial Intelligence*, Vol. 37, Nos. 1–2.
- Ehrler, L. and Cranefield, S. (2004) 'Executing agent UML diagrams', *Autonomous Agents and Multiagent Systems, Third International Joint Conference, AAMAS 2004, Proceedings*, IEEE Computer Society, pp.906–913.
- FIPA Modeling Technical Committee (2003) *FIPA Modeling: Interaction Diagrams, First Proposal, 2 July 2003*.

- FIPA Technical Committee (2002) *FIPA ACL Message Structure Specification*, 6 December 2002.
- Fu, X., Bultan, T. and Su, J. (2004a) 'Analysis of interacting BPEL web services', in S.I. Feldman, M. Uretsky, M. Najork and C.E. Wills (Eds.) *World Wide Web, Thirteenth International Conference, WWW 2004, Proceedings*, ACM, pp.621–630.
- Fu, X., Bultan, T. and Su, J. (2004b) 'Model checking XML manipulating software', *SIGSOFT Softw. Eng. Notes*, Vol. 29, No. 4, pp.252–262.
- Gardner, T. (2003) 'UML modelling of automated business processes with a mapping to BPEL4WS', *First European Workshop on Object Orientation and Web Services, EOOWS 2003, Proceedings*.
- Gomez-Sanz, J. and Pavon, J. (2003) 'Agent oriented software engineering with INGENIAS', in V. Marik, J.P. Muller and M. Pechoucek (Eds.) *Multi-Agent Systems and Applications III, Third International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Proceedings, Volume 2691 of Lecture Notes in Computer Science*, Springer, pp.394–403.
- Gomez-Sanz, J.J. and Pavon, J. (2005) 'Implementing multi-agent systems organizations with INGENIAS', in R.H. Bordini, M. Dastani, J. Dix and A.E. Fallah-Seghrouchni (Eds.) *Programming Multi-Agent Systems, Third International Workshop, ProMAS 2005, Revised and Invited Papers, Volume 3862 of Lecture Notes in Computer Science*, Springer, pp.236–251.
- Greenwood, D. and Calisti, M. (2004) 'Engineering web service-agent integration', *IEEE Systems, Cybernetics and Man Conference, Proceedings*, IEEE, Vol. 2, pp.1918–1925.
- Grønmo, R. and Solheim, I. (2004) 'Towards modeling web service composition in UML', in S. Bevinakoppa and J. Hu (Eds.) *Web Services: Modeling, Architecture and Infrastructure, Second International Workshop, WSMAI2004, Proceedings*, INSTICC Press, pp.72–86.
- Gungui, I., Martelli, M. and Mascardi, V. (2005) 'DCaseLP: a prototyping environment for multilingual agent systems', *Technical Report*, DISI, University of Genova, Italy, DISI-TR-05-20.
- Hinz, S., Schmidt, K. and Stahl, C. (2005) 'Transforming BPEL to Petri Nets', in W.M.P. van der Aalst, B. Benatallah, F. Casati and F. Curbera (Eds.) *Business Process Management, Third International Conference, BPM 2005, Proceedings*, Vol. 3649, pp.220–235.
- Holzmann, G.J. (Ed.) (2003) *The SPIN MODEL CHECKER – Primer and Reference Manual*, Addison-Wesley.
- Huget, M-P. (2002) 'Generating code for agent UML sequence diagrams', *Technical Report*, Department of Computer Science, University of Liverpool, UK, ULCS-02-020.
- Juan, T., Pearce, A.R. and Sterling, L. (2002) 'ROADMAP: extending the Gaia methodology for complex open systems', *Autonomous Agents & Multiagent Systems, First International Joint Conference, AAMAS 2002, Proceedings*, ACM, pp.3–10.
- Koning, J. and Romero-Hernandez, I. (2003) 'Generating machine processable representations of textual representations of AUML', in F. Giunchiglia, J. Odell and G. Weiß (Eds.) *Agent-Oriented Software Engineering, Third International Workshop, AOSE 2002, Revised Papers and Invited Contributions, Volume 2585 of Lecture Notes in Computer Science*, Springer, pp.126–137.
- Kummer, O., Wienberg, F., Duvigneau, M., Schumacher, J., Kohler, M., Moldt, D., Rolke, H. and Valk, R. (2004) 'An extensible editor and simulation engine for Petri Nets: Renew', in J. Cortadella and W. Reisig (Eds.) *Applications and Theory of Petri Nets, Twenty-fifth International Conference, ICATPN 2004, Proceedings, Volume 3099 of Lecture Notes in Computer Science*, Springer, pp.484–493.
- Li, Y.H., Paik, H-Y., Benatallah, B. and Benbernou, S. (2006) 'Formal consistency verification between BPEL process and privacy policy', *Privacy Security Trust, PST 2006, Proceedings*, McGraw Hill, pp.212–223.

- Luck, M., McBurney, P., Shehory, O., Willmott, S. and the AL Community (Eds.) (2005) *Agent Technology: Computing as Interaction – A Roadmap for Agent-Based Computing*, AgentLink III.
- Mantell, K. (2005) 'From UML to BPEL – model driven architecture in a web services world', <http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/>.
- Mascardi, V., Martelli, M. and Sterling, L. (2004) 'Logic-based specification languages for intelligent software agents', *Theory and Practice of Logic Programming*, Vol. 4, No. 4, pp.429–494.
- McIlraith, S.A. (2004) 'Towards declarative programming for web services', in R. Gia-cobazzi (Ed.) *Static Analysis, 11th International Symposium, SAS 2004, Proceedings, Volume 3148 of Lecture Notes in Computer Science*, Springer, p.21.
- OASIS WSBPEL Technical Committees (2006) *Web Services Business Process Execution Language (WS-BPEL) Version 2.0, 17 May 2006*.
- Object Management Group (2005a) *MOF 2.0/XMI Mapping Specification, Version 2.1, 1 September 2005*.
- Object Management Group (2005b) *Unified Modeling Language (UML): Superstructure, Version 2.0, 4 July 2005*.
- Padgham, L. and Winikoff, M. (2002) 'Prometheus: a methodology for developing intelligent agents', *Autonomous Agents & Multiagent Systems, First International Joint Conference, AAMAS2002, Proceedings*, ACM, pp.37–38.
- Provost, W. (2003) 'UML for web services', <http://www.xml.com/pub/a/ws/2003/08/05/uml.html>.
- Purvis, M., Cranefield, S., Nowostawski, M. and Carter, D. (2002) 'Opal: a multi-level infrastructure for agent-oriented software development', Discussion Paper 2002/01, Department of Information Science, University of Otago, New Zealand, http://www.otago.ac.nz/informationscience/publctns/complete/papers/dp20_02-01.pdf.gz.
- Rao, J., Kungas, P. and Matskin, M. (2004) 'Logic-based web services composition: from service description to process model', *IEEE International Conference on Web Services, ICWS'04, Proceedings*, IEEE Computer Society, pp.446–453.
- Romero-Hernandez, I. and Koning, J-L. (2004) 'A visual tool for the modeling of AXML specifications', in J. Debenham (Ed.) *Symposium on Professional Practice in AI, IFIP World Computer Congress, WCC 2004, Proceedings*, Kluwer.
- Rooney, C., Collier, R.W. and O'Hare, G.M.P. (2004) 'VIPER: a Visual Protocol Editor', in R.D. Nicola, G.L. Ferrari and G. Meredith (Eds.) *Coordination Models and Languages, Sixth International Conference, COORDINATION 2004, Proceedings, Volume 2949 of Lecture Notes in Computer Science*, Springer, pp.279–293.
- Sadri, F. and Toni, F. (1999) 'Computational logic and multi-agent systems: a roadmap', *Technical Report*, Department of Computing, Imperial College, London.
- Sterling, L. and Shapiro, E.Y. (1994) *The Art of PROLOG*, MIT Press, ISBN 0-262-19338-8.
- Sturm, A., Dori, D. and Shehory, O. (2003) 'Single model method for specifying multiagent systems', in *Autonomous Agents & Multiagent Systems, Second International Joint Conference, AAMAS2003, Proceedings*, ACM, pp.121–128.
- Thang, X. and Kowalczyk, R. (2005) 'Enabling agent-based management of web services with WS2JADE', *2005 NASA/DoD Conference on Evolvable Hardware, EH 2005, Proceedings*, IEEE Computer Society, pp.407–412.
- Van Breugel, F. and Koshkina, M. (2006) 'Models and verification of BPEL', <http://www.cse.yorku.ca/~franck/research/drafts/>.
- Verbeek, H.M.W. and van der Aalst, W.M.P. (2005) 'Analyzing BPEL processes using Petri nets', in D. Marinescu (Ed.) *Applications of Petri Nets to Coordination, Workflow and Business Process Management, Second International Workshop, Proceedings*, pp.59–78.
- W3C Web Services Choreography Working Group (2002) *Web Service Choreography Interface (WSCI) Version 1.0, 8 August 2002*.

- W3C Working Group (1999) *XSL Transformations (XSLT) Version 1.0*, 16 November 1999.
- Wagner, G. (2004) 'AOR modelling and simulation towards a general architecture for agent-based discrete event simulation', *Agent-Oriented Information Systems, Fifth International Bi-Conference Workshop, AOIS 2003, Revised Selected Papers, Volume 3030 of Lecture Notes in Computer Science*, Springer, pp.174–188.
- Walton, C. (2005) 'Uniting agents and web services', *AgentLinkNews*, Vol. 18, pp.26–28.
- Web Services Description Working Group (2006) *Web Services Description Language (WSDL) Version 2.0*, 27 March 2006.
- Winikoff, M. (2005) 'Towards making agent UML practical: a textual notation and a tool', *2005 NASA/DoD Conference on Evolvable Hardware, EH 2005, Proceedings*, IEEE Computer Society, pp.401–412.
- Zambonelli, F., Jennings, N.R. and Wooldridge, M. (2003) 'Developing multiagent systems: the Gaia methodology', *ACM Transactions on Software Engineering and Methodology*, Vol. 12, No. 3, pp.317–370.

Notes

- 1 <http://www.cs.rmit.edu.au/agents/protocols/>
- 2 <http://lia.deis.unibo.it/research/socs/>
- 3 <http://centria.di.fct.unl.pt/~clima/>
- 4 <http://staff.science.uva.nl/~ulle/DALT-2006/home.html>
- 5 A recent survey of models and verification tools for BPEL is provided by van Breugel and Koshkina (2006).
- 6 <http://www.disi.unige.it/person/MascardiV/Software/WEST2EAST.html>
- 7 <http://www.jdom.org/>
- 8 <http://www.cs.rmit.edu.au/agents/pdt/>
- 9 <http://www.agent-software.com>
- 10 Consider for example the two tutorials <http://www.w3schools.com/wSDL/default.asp> and http://www.eclipse.org/stp/b2j/docs/tutorials/wsbpel/wsbpel_tut.php, dealing with WSDL and WS-BPEL, respectively.
- 11 The reader may consider one of the online Prolog tutorials, such as <http://www.coli.uni-saarland.de/~kris/learn-prolog-now/lpnpag.php?pageid=online>, besides the classic book *The Art of Prolog* by Sterling and Shapiro (1994).
- 12 Note that here, `sFinal` is a Prolog variable that stands for any Prolog term representing the state to reach at the end of the cycle; it should not be confused with the `final` constant that represents the state in which the protocol terminates.
- 13 <http://www.mysql.com/>
- 14 <http://www.alphaworks.ibm.com/tech/ettkws>
- 15 <http://www.parasoft.com/jsp/products/home.jsp?product=BPEL&itemId=114>
- 16 <http://www.306.ibm.com/software/websphere>
- 17 <http://www.oracle.com/technology/products/ias/bpel/index.html>
- 18 <http://ingenias.sourceforge.net/>
- 19 <http://www.cl.cam.ac.uk/research/hvg/Isabelle/index.html>