From AUML to WS-BPEL

Giovanni Casella and Viviana Mascardi
DISI, Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy
casella@disi.unige.it, mascardi@disi.unige.it

Abstract

The Web Services (WS) technology is currently gaining a wider and wider consensus. The features that characterise WSs, namely heterogeneity, distribution, openness, highly dynamic interactions, are some among the key characteristics of another emerging technology, that of intelligent agents and Multi-Agent Systems (MASs).

In this paper we discuss the relationships between WSs and intelligent agents and we propose our point of view, namely that agents provide both the coordination framework and the engineering metaphor that can be exploited for realising complex applications based on the WSs infrastructure. Based on our claim, we suggest to use an agent-oriented extension of UML 2.0 named AUML to model agent interaction protocols, and a business protocol execution language for WSs named WS-BPEL, to publish the specification of these protocols on the Web. To demonstrate the feasibility of our approach, we have designed and implemented a tool that automatically creates WS-BPEL and WSDL specifications of interaction protocols starting from AUML visual diagrams.

1. Introduction and Motivation

Thanks to its applicability to many heterogeneous domains, the Web Services (WS) technology is gaining a wider and wider consensus. According to the W3C (18), "*a WS is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialisation in conjunction with other Web-related standards.*"

Software applications written in various programming languages and running on various platforms can both expose themselves as WSs, and use other WSs. Thus, WSs are heterogeneous in their nature. Also, they are designed to allow machine-to-machine interaction. This interaction takes place over a network, such as the Internet, thus WSs are by definition distributed, and operate in an open and highly dynamic environment.

Heterogeneity, distribution, openness, highly dynamic interactions, are some among the key characteristics of another emerging technology, that of intelligent agents and Multi-Agent Systems (MASs). The AgentLink III Agent Technology Roadmap (22) proposes the following definition: "*an agent is a computer system that is capable of flexible autonomous action in dynamic, unpredictable, typically multi-agent domains.* "

WSs and intelligent software agents share many common features, and this suggests that some relationship between the two technologies should exist. Actually, the most recent literature in the agents' field devotes much space to these relationships.

Also limiting ourselves to publications of the last year (2005), we find many considerations on the links between agents and WSs. For example, the already cited AgentLink III Roadmap puts in evidence that WSs "*provide a ready-made infrastructure that is almost ideal for use in supporting agent interactions in a multi-agent system. More importantly, perhaps, this infrastructure is widely accepted, standardised, and likely to be the dominant base technology over the coming years. Conversely, 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.*"

A very nice representation of the key agent techniques, and their approximate WS equivalents, has been given by C. Walton in (31), where agents have their counterpart in WSs, agent communication is related to SOAP and WSDL, and agent ontologies are related to RDF-S and OWL (Figure 1). In that paper, Walton proposes to decompose agents into a stub, that executes agent interaction protocols and is responsible for communication between agents, and a body, which encapsulates the reasoning processes, and is encoded as a set of decision procedures. Both the stub and the body should be implemented as WSs.

In (9), P. A. Buhler and J. M. Vidal summarise the relationship between agents and WSs with the aphorism "*Adap-*

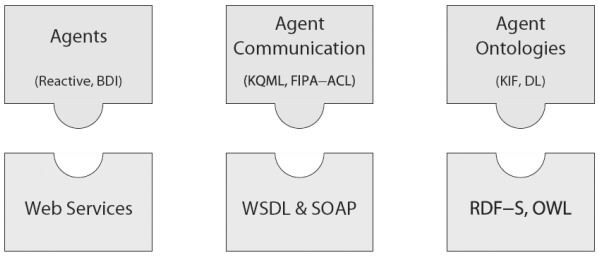

Figure 1. Contrasting WS technologies and their analogies with MAS techniques, from (31)

tive Workflow Engines = Web Services + Agents": namely, WSs provide the computational resources and agents provide the coordination framework. They propose the use of the BPEL4WS language as a specification language for expressing the initial social order of the multi-agent system.

The CooWS architecture defined by L. Bozzo, V. Mascardi, D. Ancona and P. Busetta (6) exploits BPEL4WS for representing the behavioural knowledge of agents modelled according to the BDI architecture (29) and able to cooperatively exchange plans ("CooBDI" agents).

The view of the cited researchers is that agents and WSs are complementary tools: WSs provide the infrastructure, and agents provide the coordination framework.

However, WSs are not always seen as mere infrastructure providers: referring to WS choreography languages, M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella (3) write that "*choreographies and agent interaction protocols undoubtedly share a common purpose. In fact, they both aim at expressing global interaction protocols, i.e. rules that define the global behavior of a system of cooperating parties*." Thus, WS and agent technologies share some commonalities also in their goal of providing tools, languages, and methods necessary for engineering systems that behave in a correct way, for example w.r.t. a given interaction protocol.

Our position integrates both points of view. We think that, besides the coordination framework, agents also provide the *engineering metaphor* that can be exploited in order to realise MASs based on the WSs infrastructure. Based on this claim, we propose to use an agent-oriented extension of UML 2.0 named Agent UML (AUML for short) to model agent interaction protocols, and a business protocol execution language for WSs named WS-BPEL, to publish the specification of these protocols on the Web. By publishing these specifications on the Web, agents can easily learn how to communicate. MAS, in fact, integrate collaborative agents that need to communicate with other agents whose features are initially unknown. By reading the WS-BPEL document published by the "publisher agent", the "reader" can learn the communication protocol to follow in order to obtain the required service.

To demonstrate the feasibility of our approach, we have designed and implemented a tool that automatically creates WS-BPEL and WSDL specifications of interaction protocols starting from AUML visual diagrams.

The idea of designing and developing a "UML2PEL" translator is not completely new. In (23), K. Mantell describes a tool which takes processes defined in UML and generates the corresponding BPEL4WS[1] and WSDL files to implement that process. The only UML diagram that could be translated into BPEL4WS (according to the paper; no further check is possible since the tool is no longer available) is the activity diagram. The proposed approach enables service-oriented BPEL4WS components to be incorporated into an overall system design utilising existing software engineering practices. Additionally, the mapping from UML to BPEL4WS permits a model-driven development approach in which BPEL4WS executable processes can be automatically generated from UML models. Thus, the motivations for the implementation of a "UML2BPEL" translator are to raise the level of abstraction at which development occurs, which, in turn, delivers greater productivity, better quality, and insulation from underlying changes in technology. Also, the approach provides an integration mechanism for multiple standards and specifications that need to be used to build a complex solution, which is especially relevant in business integration scenarios.

To the best of our knowledge, apart from the software described in (23), no other proposals of exploiting UML for engineering WSs exist. Our approach is complementary to K. Mantell's one: we translate interaction protocols represented as *AUML protocol diagrams* into *WS-BPEL abstract processes*, and he translates *UML activity diagrams* into *executable BPEL4WS processes*. Besides this, Mantell's contribution is mainly concerned with the engineering stage, while our contribution also applies to the deployement stage. In fact, once the WS-BPEL specifications of the interaction protocols have been published on the Web, any agent able to read a WS-BPEL document can take advantage of them for learning how to interact with the publisher of the document.

The paper is organised in the following way: Sections 2 and 3 introduce AUML and the technologies behind WSs, respectively. Section 4 discusses our approach to engineering agent interaction protocols exploiting AUML for the modelling stage, and WS-BPEL and WSDL for publishing the protocol specification on the Web. Conclusions and future work are discussed in Section 5.

1 BPEL4WS is the name of the previous version of WS-BPEL.

2. Agent Interaction Protocols in AUML

Agent-Oriented Software Engineering (AOSE (28; 16; 17; 34; 12)) studies how existing techniques can be adapted or extended in order to engineer MASs. Currently, many AOSE methodologies (for example, Gaia (35)), notations (such as AUML (4; 19)) and projects (the "Flexible and Robust Protocol-Based Interaction between Agents in Open Systems[2]" and "Societies Of ComputeeS (SOCS)[3]", just to cite two recent ones) take *Agent Interaction Protocols (AIPs)* as their starting point. We consider the AUML interaction diagrams specification proposed by the FIPA Technical Modeling Committee (20) as our reference notation for AIPs.

An AIP is usually defined in relation to a particular agent activity. AIPs in AUML extend UML 2.0 sequence diagrams (27). The protocol name is contained in a box on the upper-left corner of the diagram and is preceded by the keyword `sd` (sequence diagram). The AIP is characterised by the following features:

- *Actors, roles, and lifelines.* An AIP consists of some lifelines; for each lifeline, a rectangular box contains information about the agent to which that lifeline belongs, the role it plays in the MAS, and its class. The syntax used is `agent-name/agent-role:agent-class`. The rectangular box can also indicate a general set of agents playing a specific role, instead of a specific instance of an agent. The syntax is already part of UML 2.0, except that the UML 2.0 syntax indicates an object name instead of an agent name.
- *Message exchange.* Labelled arrows between two lifelines represent messages. Messages must satisfy standardised communicative (speech) acts which define the type and the content of the messages (e.g. the FIPA agent communication language, FIPA-ACL (15), or KQML (24)). Since a protocol that has a single, sequential path from the initial state to the final state is worthless and reduces the autonomy for agents, AUML allows agents to consider several alternatives during the interaction based on their mental states, their intentions and the current state of the interaction. To this aim, it supports different interaction operators. A box, with an interaction operator given in its top-left corner, can surround a part of the sequence diagram. Boxes can recursively contain messages and other boxes, and can be divided into a number of regions separated from each other by heavy horizontal dashed lines. Each region can include a condition depicted as text in square brackets. If the condition is satisfied, then that region is selected and the activities inside it are performed. The interaction operators supported by AUML include `weak sequencing`, `alternative`, `option`, `parallel` and `loop` (see (20)).
- *Weak sequencing operator.* The `weak sequencing` operator as defined in UML 2.0 means a weak sequencing in a sequence of messages, that is to say, the messages within this box on the same lifeline are ordered but it is not possible to make any assumption for message ordering coming from different lifelines in the same box.
- *Alternative operator.* `Alternative` means that there are several paths to follow, and that agents have to choose at most one to continue. Guards are associated with alternatives: when a guard is evaluated to true, that alternative path is chosen. An `else` alternative may be optionally present: it is entered if no guard evaluates to true. If no guard is evaluated to true, and no `else` alternative is provided, the `alternative` box of the diagram is not executed. Alternatives were handled through the XOR operator in previous AUML versions.
- *Option operator.* The `option` operator only considers one path in the region. If the condition associated with this path is evaluated to true, then the path is executed, else nothing happens and the interaction follows after this portion of the diagram. The `option` operator corresponds to the OR operator with just one alternative of previous AUML versions.
- *Parallel operator.* The `parallel` operator depicts the parallel execution of different paths in any order. It allows designers to represent the delivery of several messages concurrently and is similar to the AND connector in previous versions of AUML.
- *Loop operator.* The `loop` operator allows designers to represent that an ordered set of messages has to be applied several times. Designers can use either lower and upper bounds or a boolean expression. As long as conditions are satisfied, the loop is executed and the messages are sent and received.

In AUML, parameters are written separately outside the diagram in a comment, stereotyped as `<<parameters>>`. AIPs can have many parameters such as content language for messages, agent communication language (ACL), ontology, etc.

To make an example, Figure 2 shows an AIP modelled using AUML. The diagram is named `JobSearch` (upper left label), and describes a scenario where a job manager (the agent on the left, named `jm`, playing the role of `jobManager` and belonging to the

2 `http://www.cs.rmit.edu.au/agents/protocols/`
3 `http://lia.deis.unibo.it/research/socs/`

jobManagerClass) interacts with an agent looking for a job (the agent on the right, named js, playing the role of jobSearcher and belonging to the jobSearcherClass).

The content language is first order logic, and we assume that an ontology named job-search contains all the information required by the agents to understand the content of the exchanged messages (the <<parameters>> label on the top of the diagram). Since it is becoming the de facto standard, we adhere to the FIPA-ACL message syntax; 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 interaction is started by the job searcher, who requests to the job manager to inform him about the available jobs that meet some condition, for example related to salary and vacations (first message from the top, with request performative and ''inform-me-about-jobs(cond)'' content).

The job manager first provides the information about the number of available jobs that meet the condition (inform(''available-job-number(num)'')), and then enters a loop (loop box) where each of the available jobs is proposed to the job searcher, together with a description of its features (cfp (''job(i, description)'')). The job searcher may either (alternative box): 1) answer that he did not understand either the ontology or the message content (not-understood (''message-content'') and not-understood (''ontology'')), after which the job searcher interrupts its participation to the interaction protocol; or 2) accept the proposal as it is (accept-proposal(''job(i, description)'')); or 3) execute the sequence of activities (weak sequencing box) that consist of refusing the proposal (reject-proposal(''job(i, description)'')), maybe because something in the job specification is not acceptable, although the initial condition is met, and requesting to the job manager to make another proposal that takes some additional conditions into account (request (''another-prop-for-job(i, add-cond)'')). In this case, the job manager makes another proposal for that job (cfp(''job(i, new-description)'')), that the job searcher may either accept or refuse (nested alternative box).

In case one proposal has been accepted by the job searcher (option box), the job manager sends the contact details of the job provider to the job searcher (inform(''contact-information-for-job(i, c)'')), and the interaction ends.

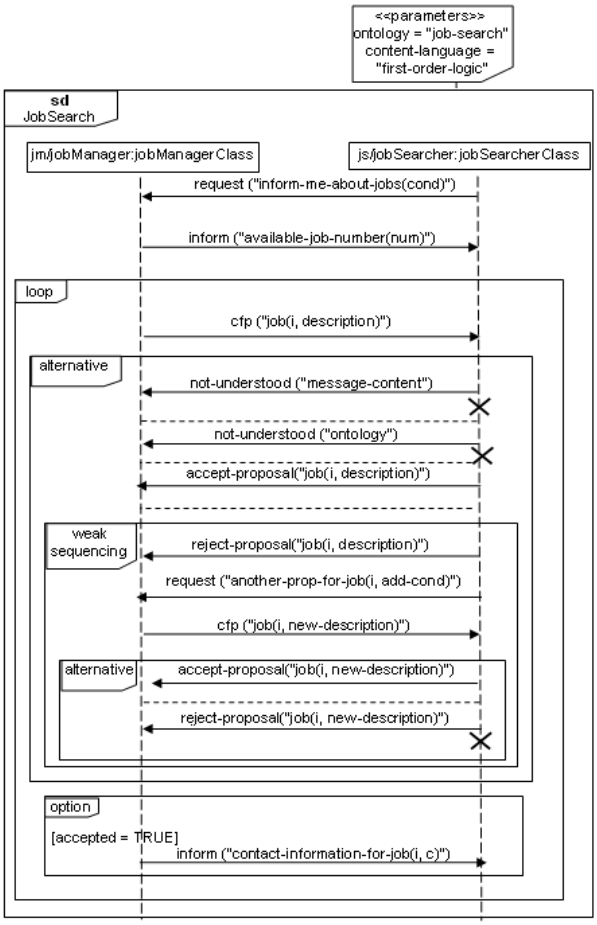

Figure 2. An interaction protocol in AUML

3. Web Services

The goal of WSs is to achieve universal interoperability between applications by using Web standards. The following basic specifications originally defined the WSs space: SOAP (Simple Object Access Protocol, (25)), WSDL (Web Services Description Language, (11)), and UDDI (Universal Description, Discovery, and Integration, (30)). All of them lie upon XML (7), and their relationships are depicted in Figure 3. WS-BPEL (once named BPEL4WS) provides a language for the formal specification of business processes and business interaction protocols. By doing so, it extends the WSs interaction model and enables it to support business transactions. Since our work is mainly based on WSDL and WS-BPEL, we will only describe them.

WSDL. WSDL defines an XML grammar for describing network services as collections of communication endpoints, or ports, capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details in-

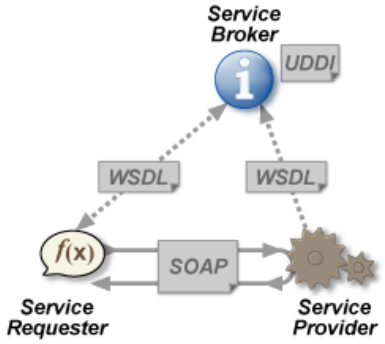

Figure 3. Standards for WSs , from (32)

volved in applications communication. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment: messages are abstract descriptions of the data being exchanged, and port types are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitute a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Hence, a WSDL document uses the following elements in the definition of network services:

- *Types*: a container for data type definitions using some type system.
- *Message*: an abstract, typed definition of the data being communicated.
- *Operation*: an abstract description of an action supported by the service.
- *PortType*: an abstract set of operations supported by one or more endpoints.
- *Binding*: a concrete protocol and data format specification for a particular port type.
- *Port*: a single endpoint defined as a combination of a binding and a network address.
- *Service*: a collection of related endpoints.

WSDL supports the XML Schemas specification (5) as its canonical type system.

WS-BPEL. The Web Services Business Process Execution Language (WS-BPEL (1)) is a notation for specifying business process behaviour based on WSs. Processes in WS-BPEL export and import functionality by using WS interfaces exclusively.

WS-BPEL has been conceived to allow the definition of business protocols. WS-BPEL uses a notion of message properties, which are a type of variable property, to identify protocol-relevant data embedded in messages. Properties can be viewed as "transparent" data relevant to public aspects as opposed to the "opaque" data that internal/private functions use. Transparent data affects the public business protocol in a direct way, whereas opaque data is significant primarily to back-end systems and affects the business protocol only by creating nondeterminism because the way it affects decisions is opaque.

A WS-BPEL process can define a business protocol role, using the notion of abstract process. The relationship between roles is typically modelled as a "partner link". Abstract processes use all the concepts of WS-BPEL but approach data handling in a way that reflects the level of abstraction required to describe public aspects of the business protocol. Specifically, abstract processes handle only protocol-relevant data, that are identified as message properties. In addition, abstract processes use nondeterministic (opaque) data values to hide private aspects of behaviour. WS-BPEL can also be used to define executable processes, but their description is out of the scope of this paper.

In the following list, we describe WS-BPEL's main features.

- *Partner Links.* The WS-BPEL partner link type characterises the conversational relationship between two services by defining the "roles" played by each service in the conversation, and specifying the `portType` provided by each service to receive messages within the context of the conversation. Each partner link is characterized by a `partnerLinkType`, and more than one partner link can be characterised by the same `partnerLinkType`. The role of the business process itself is indicated by the attribute `myRole` and the role of the partner is indicated by the attribute `partnerRole`. In the degenerate case where a `partnerLinkType` has only one role, one of these attributes is omitted as appropriate.
- *Message exchange.* Message exchange can be implemented by performing the `invoke` and `receive` activities that WS-BPEL offers. Invoking an operation on a service provided by a partner (specified by the partner link) can be either a synchronous request/response operation or an asynchronous one-way operation. WS-BPEL uses the same basic syntax for both with some additional options for the synchronous case. An asynchronous invocation requires only the input variable of the operation because it does not expect a response as part of the operation. A synchronous invocation requires both an input variable and an output variable. In the case of a synchronous invocation, the operation might return a WSDL fault message that results into a WS-BPEL fault. Such a fault can be caught locally by the activity, or thrown to the scope that encloses the activity.

- *Sequence activity.* A sequence activity contains one or more activities that are performed sequentially, in the order in which they are listed within the `sequence` token. The sequence activity completes when the final activity in the sequence has completed.
- *Switch activity.* The `switch` activity consists of an ordered list of one or more conditional branches defined by `case` elements, followed optionally by an `otherwise` branch. The `case` branches of the `switch` are considered in the order in which they appear. The first branch whose condition holds true is taken and provides the activity performed for the switch. If no branch with a condition is taken, then the `otherwise` branch is taken (if not explicitly specified, then an `otherwise` branch with an empty activity is deemed to be present). The `switch` activity is complete when the activity of the selected branch completes.
- *If activity.* The `if` activity consists of an ordered list of one or more conditional branches defined by `if`, `elseif` elements, followed optionally by an `else` branch. The `if` and `elseif` branches of the `if` activity are considered in the order in which they appear.
- *Flow construct.* The `flow` construct provides concurrency and synchronisation. The most fundamental semantic effect of grouping a set of activities in a flow is to enable concurrency. A `flow` completes when all of the activities in the flow have completed. Completion of an activity in a flow includes the possibility that it will be skipped if its enabling condition turns out to be false. Thus the simplest use of `flow` is equivalent to a nested concurrency construct.
- *While activity.* The `while` activity supports repeated performance of a specified iterative activity, performed until the given boolean `while` condition no longer holds true.

WS-BPEL is layered on top of several XML specifications: WSDL 1.1, XML Schema 1.0 (5), and XPath 1.0 (14). WSDL messages and XML Schema type definitions provide the data model used by WS-BPEL processes. XPath provides support for data manipulation. All external resources and partners are represented as WSDL services. WS-BPEL provides extensibility to accommodate future versions of these standards, specifically the XPath and related standards used in XML computation.

4. From AUML to WS-BPEL

Many of the features that characterise WS-BPEL abstract processes, make it very suitable to represent AIPs and correspond in a pretty precise way to those that characterise AUML (Table 1). Due to this tight correspondence

	AUML	**WS-BPEL**
Roles	`ag-name/ ag-role: ag-class` box on top of lifelines	`myRole` and `partnerRole` tags in Partner Links
Message	Labelled arrows between lifelines	`invoke` and `receive` act.
Content	Speech-act based	Unspecified
Seq.	`Weak Sequencing` op.	`Sequence` act.
Cond.	`Alternative` op.	`Switch` act.
Option	`Option` op.	`If` act.
Concurr.	`Parallel` op.	`Flow` act.
Cycle	`Loop` op.	`While` act.

Table 1. Correspondence between AUML and WS-BPEL concepts

and to the existing relationships between agents and WSs introduced in Section 1, we propose to use AUML for representing interaction protocols and WS-BPEL and WSDL for publishing them on the Web. This section describes the ideas behind the automatic translator from AUML to WS-BPEL.

From AUML visual diagrams to a textual notation

Visual diagrams have been conceived and designed for human beings, not for machines, thus it is very difficult to reason and work on them for computers.

In order to be processed in an automatic way by computers, textual notations are still widely considered to have some significant advantages. For example, M. Winikoff (33) observes that defining the syntax of a textual notation is considerably simpler than defining the precise syntax of a two-dimensional graphical notation.

For this reason, we use a textual notation for AUML interaction diagrams, in order to automatically generate a couple of WSDL and WS-BPEL specifications compliant with the AUML visual diagram. We extend the notation proposed by Winikoff in (33) by adding some more elements to it, and by putting some more constraints. For our convenience, we also add some "syntactic sugar" to Winikoff's notation, in order to make it XML-based.

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`): the shortname is used to refer to the agent when sending mes-

sages whereas the longname is used in the box at the top of the lifeline. This avoids having to repeatedly type the long agent name in messages.
– Messages between agents (`message`)
– The start and end of boxes (`box` and `end`)
– The type of the box (`weak sequencing`, `alternative`, `option`, etc)
– The boundary between regions within a box (`next`) and guards (`guard`)
– Continuations (`goto` and `label`)
– 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`

Besides this, we assume that in our interaction diagrams there are always two agents: one that publishes the document that specifies how the interaction should be carried on (`agent-publisher`), and one that reads this document (`agent-reader`). The message exchange is compliant with FIPA-ACL.

Finally, separators and terminal symbols in our notation are XML tags instead of newlines and keywords, as in Winikoff's one.

From the textual notation to WS-BPEL

If we put some conditions on the structure of the AUML protocol diagram (and, as a consequence, of its textual representation), we can define a translation of AUML protocol diagrams into WS-BPEL. In particular, we need to establish once and for all which agent, between two (and exactly two!) depicted in the AUML diagram, is the publisher of the WSDL and WS-BPEL documents, which one is the reader, and which agent starts the communication. We assume that the agent with the leftmost lifeline is the publisher, and that the communication protocol starts with the publisher receiving a message.

WSDL. We start with the WSDL document that describes the data structures used by the WS-BPEL process. The WSDL document begins with standard definitions of name spaces, such as

```
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
```

After them, the seven types used in the document are defined:

1. *FIPA-ACL performatives*. For each performative, an element of type string, whose value is fixed to the name of the performative, is defined:

```
<xs:element name="agree_ca" type="xs:string" fixed="agree"/>
<xs:element name="cfp_ca" type="xs:string" fixed="cfp"/>
```

The definition of FIPA-ACL performatives does not depend on the AUML protocol diagram under translation: it is always the same for all diagrams.

2. *Protocol diagram mandatory parameters.* The ontology and content language names read from the `<<parameters>>` comment in the AUML diagram, as well as the protocol name, are defined. From now on, the translation becomes dependent on the input diagram. In the following, we will refer to Figure 2 as our running example.

```
<xs:element name="ontology_name"
    type="xs:string" fixed="job-search"/>
<xs:element name="content_language_name"
    type="xs:string" fixed="first-order-logic"/>
<xs:element name="protocol_name"
    type="xs:string" fixed="sd JobSearch"/>
```

3. *Information on the reader and on the publisher.* The name and role of the agents that partecipate to the protocol are read from the AUML diagram. We recall our assumption that the publisher of the document is the leftmost agent in the AUML diagram.

```
<xs:element name="publisher_name"
    type="xs:string" fixed="jm"/>
<xs:element name="publisher_role"
    type="xs:string" fixed="jobManager"/>
<xs:element name="reader_name"
    type="xs:string" fixed="js"/>
<xs:element name="reader_role"
    type="xs:string" fixed="jobSearcher"/>
```

4. *Communication links.* Using complex types, the "Participant_MsgFromPublisher" and the "Participant_MsgFromReader" are defined. They are characterised by the sender (publisher, or, resp. reader), the receiver (reader or, resp. publisher) and reply_to field, that coincides with the sender.

5. *Variables.* For each alternative, option and loop box in the AUML diagram, a type with a different structure is defined. In the WS-BPEL document, these types will be assigned to opaque variables that will be instantiated by either the reader or the publisher, and whose value will determine which one among the possible execution paths, will be followed. For example, for the first alternative box in the diagram depicted in Figure 2, characterised

by four alternatives, the following "choose" type is defined:

```
<xs:element name="choose_type" >
  <xs:simpleType><xs:restriction base="xs:integer">
    <xs:minInclusive value="1" />
    <xs:maxInclusive value="4" />
  </xs:restriction></xs:simpleType>
</xs:element>
```

6. *Messages.* For each message exchanged, a message type is defined. It is characterised by the message performative, the communication link (either "Participant_MsgFromReader" or "Participant_MsgFromPublisher"), the message content (an unspecified string), the content language and ontology, as they have been defined in the "content_language_name" and "ontology_name" elements, and the protocol name defined by the "protocol_name" element. For example, the type corresponding to the first message in the AUML protocol diagram of our example is:

```
<message name="Mess_1">
  <part name="performative" element="request_ca"/>
  <part name="Participant"
    type="tns:Participant_MsgFromReader"/>
  <part name="Content" element="tns:message_content"/>
  <part name="Content_Language"
    element="tns:content_language_name"/>
  <part name="Ontology" element="tns:ontology_name"/>
  <part name="Protocol" element="tns:protocol_name"/>
</message>
```

7. *Port Types.* The last types defined in the WSDL document are port types, one for the publisher ("publisherPT") and one for the reader ("readerPT"). Each port type defines the operation types performed by the publisher (resp. the reader), that may be either reception or delivery of messages. The received (sent) messages are also defined. The first two operations declared in the publisher port type, for example, are:

```
<portType name="publisherPT">
  <operation name="RCV_Mess_1">
    <input message="tns:Mess_1"/>
  </operation>
  <operation name="SND_Mess_2">
    <output message="tns:Mess_2"/>
  </operation>
```

The WSDL document ends with the definition of the partner link types:

```
<plnk:partnerLinkType name="jobManagerjobSearcherLinkLT">
  <plnk:role name="jobManager">
    <plnk:portType name="tns:publisherPT"/>
  </plnk:role>
  <plnk:role name="jobSearcher">
    <plnk:portType name="tns:readerPT"/>
  </plnk:role>
</plnk:partnerLinkType>
```

WS-BPEL. The WS-BPEL document is based on the WSDL one. After some initial standard declarations, the WS-BPEL document begins with the partner links definitions, that refer to the partner link types in the WSDL document:

```
<partnerLinks>
  <partnerLink name="publisherPL"
    partnerLinkType="lns:jobManagerjobSearcherLinkLT"
    myRole="jobManager" partnerRole="jobSearcher"/>
  <partnerLink name="readerPL"
    partnerLinkType="lns:jobSearcherjobManagerLinkLT"
    myRole="jobSearcher" partnerRole="jobManager"/>
</partnerLinks>
```

The roles that appear in the partner links are those defined in the rectangular boxes on top of the lifelines, in the AUML diagram.

For each variable type defined in the WSDL, a variable is declared. After this, the translation of the AUML operators into WS-BPEL activities takes place. The translation is driven by the correspondence between AUML operators and WS-BPEL activities depicted in Table 1. For example, the first two messages exchanged in the diagram of Figure 2 are not in the scope of any AUML operator. They are just in sequence, and their translation, that refers to the operations defined in the WSDL port types, is thus:

```
<receive partnerLink="publisherPL" portType="lns:publisherPT"
  operation="RCV_Mess_1" createIstance="yes"/>
<invoke partnerLink="readerPL" portType="lns:readerPT"
  operation="SND_Mess_1"/>
<receive partnerLink="readerPL" portType="lns:readerPT"
  operation="RCV_Mess_2"/>
<invoke partnerLink="publisherPL" portType="lns:publisherPT"
  operation="SND_Mess_2"/>
```

Note that both the point of view of the publisher (that receives the first message and sends the second one), and that of the receiver (that sends the first message and receives the second one), are considered in the WS-BPEL document. Since the document is meant to be published on the web, and any agent may read it and adhere to the protocol that it defines, the points of view of both participants to the protocol must be taken into account.

After having assigned an opaque value to the variable "continue_1", it is used in the guard of the "while" activity that corresponds to the "loop" box in the AUML diagram:

```
<while condition="$continue_1.value=true">
    <receive partnerLink="readerPL" portType="lns:readerPT"
        operation="RCV_Mess_3"/>
    <invoke partnerLink="publisherPL" portType="lns:publisherPT"
        operation="SND_Mess_3"/>
```

A "switch" activity, nested into the "while" one, corresponds to the "alternative" box in the AUML diagram. The value of the "choose2" variable (with type "choose_type" defined in the WS-BPEL document) ranges in [1 .. 4]. We only show the first statements of the fourth alternative, where a "weak sequencing" box, translated into a "sequence" activity, appears.

```
<switch>
    <case condition="$choose2.value=1">
        <sequence>
            <receive partnerLink="publisherPL"
             portType="lns:publisherPT" operation="RCV_Mess_4"/>
            .... </sequence> </case>
```

Another "switch" activity, nested into this one, represents the second "alternative" box in the AUML diagram.

The last activity that we find in the WS-BPEL document is an "if-then", that translates the "option" box in the AUML diagram. Also in this case, the variable "condition_5" is opaque.

```
<if> <condition condition="$condition_5.value=true"/>
    <then>
        <receive partnerLink="readerPL"
            portType="lns:readerPT" operation="RCV_Mess_11"/>
        <invoke partnerLink="publisherPL"
            portType="lns:publisherPT" operation="SND_Mess_11"/>
    </then></if>
```

Implementation

We have implemented an "AUML2WS-BPEL" translator that takes an AUML visual AIP and generates the WSDL and WS-BPEL documents that correspond to it, using our XML-based notation for AUML as intermediate format. Since currently no editors for AUML exist, we limited ourselves to consider only those features that AUML shares with UML 2.0, and only a subset of them, in order to edit the AUML AIPs with existing tools for UML.

Our translator, developed in Java, takes as input the XMI (26) description of a UML 2.0 sequence diagram (corresponding to an AUML interaction protocol). By exploiting the Extensible Stylesheet Language Transformations (XSLT) technology (13), the features of the XMI document relevant for generating the AUML textual notation are extracted and both a document in Winikoff's textual notation,

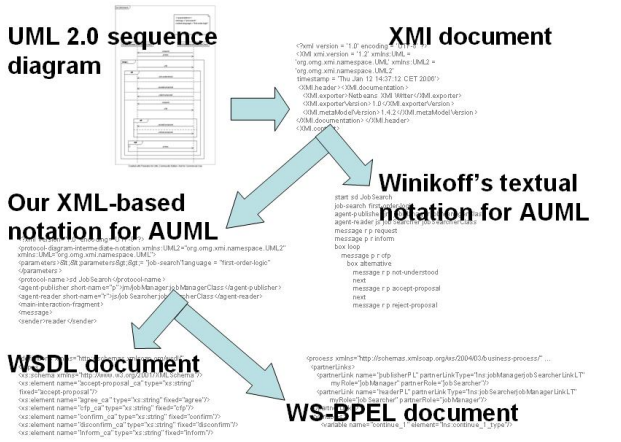

Figure 4. Translation process

and another one in our XML-based notation, are generated. The XML-based document is used by another XSLT sheet, to generate both the WSDL and the WS-BPEL documents. Figure 4 graphically shows the automatic translation process.

We have tested the translator drawing the UML 2.0 sequence diagrams with Poseidon[4], and exporting them into XMI. It should be possible to use any other UML editor supporting the 2.0 version. The WSDL and WS-BPEL code inserted in this section has been generated with our translator, freely available to the research community from `http://www.disi.unige.it/person/MascardiV/Software/AUML2WS-BPEL.html`.

5. Conclusions and Future Work

In this paper we have described the motivations, the design and the implementation of an AUML to WS-BPEL translator. Although its main purpose is to generate the WSDL and WS-BPEL specifications corresponding to a given AUML AIP, the translator that we have developed also produces a textual specification of the AUML visual diagram that respects Winikoff's grammar, and a specification that "XML-ises" (adds some XML to...) Winikoff's formalism, and that is used as internal intermediate format.

When we started to face the problem of moving from AUML to the WSs world, we considered both the Web Services Choreography Description Language (WS-CDL, (21)) and the Web Service Choreography Interface (WSCI, (2)) as target languages of the translation process, instead of WS-BPEL. WS-CDL and WSCI are about choreography, that describes the system in a top-down manner, while WS-BPEL is about orchestration, that focusses on single peers

4 `http://gentleware.com/index.php`

description. WS-CDL, although successive to WSCI, offers less operators than WSCI. On the other hand, WSCI, although richer that WSC-DL, is no longer maintained. Instead, WS-BPEL provides a rich set of constructs and is continuously maintained. Besides this, the relationships between the activities that WS-BPEL offers and AUML operators, are very natural. This is not equally true for the relationships between WS-CDL constructs and AUML ones. For these reasons, we opted for WS-BPEL.

The choice of considering an orchestration language for representing the agent's AIPs is not the only possible one. Although we are not aware of projects for translating UML or AUML specifications into WS-CDL or WSCI, many researchers working in the agent field are currently exploring the advantages of using a choreography language to describe the rules that agents should respect in their interaction. For example, M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella (3) propose an approach to the verification of a priori conformance of a business process to a protocol, which is based on the theory of formal languages and guarantees the interoperability of agents that are individually proved to conform to the protocol.

In (8), A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo formalise the WSCI language 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 N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, that define a notion of conformance between choreography and orchestration which allows to state when an orchestrated system conforms to a given choreography. Choreography and orchestration are formalised by using two process algebras and conformance takes the form of a bisimulation-like relation (10).

Our short term work involves the improvement of the AUML2WS-BPEL translator. In fact, its debugging has not been completed yet, and, for the moment, it is able to translate only `option`, `seq`, `loop`, and `alt` UML operators into the corresponding `if-then`, `sequence`, `while` and `switch` WS-BPEL activities. After the debugging will be completed, and the set of translated UML operators will be richer than now, we plan to develop a MAS where agents produce and use the WSDL and WS-BPEL specifications in order to engage a conversation. This would allow us to demonstrate that the translation from AUML to WS-BPEL and WSDL is not only possible, but also useful.

In the medium/long term, WS choreography languages will be taken into account, with the purpose of defining a unified approach that captures both global and individual aspects of the interaction protocols, and that integrates our results with the results obtained by other researchers.

References

[1] A. Arkin, S. Askary, B. Bloch, F. Curbera, Y. Goland, N. Kartha, C. K. Liu, S. Thatte, P. Yendluri, and A. Yiu, editors. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, 2005. Oasis Committee Draft 01 September 2005 downloadable from `http://www.oasis-open.org/committees/download.php/14616/wsbpel-specification-draft.htm`.

[2] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P. Takacsi-Nagy, I. Trickovic, and S. Zimek, editors. *Web Service Choreography Interface (WSCI) version 1.0*. W3C Note 8 August 2002 `http://www.w3.org/TR/wsci`.

[3] M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying the conformance of web services to global interaction protocols: A first step. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *Proceedings of the Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop (EPEW 2005) and International Workshop on Web Services and Formal Methods, (WS-FM 2005)*, pages 257–271. Springer-Verlag, 2005.

[4] B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. In Ciancarini and Wooldridge (12), pages 91–104.

[5] D. Beech, N. Mendelsohn, M. Maloney, and H. S. Thompson, editors. *XML Schema Part 1: Structures (Second Edition)*, 2004. W3C Recommendation 28 October 2004 downloadable from `http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/`.

[6] L. Bozzo, V. Mascardi, D. Ancona, and P. Busetta. CooWS: Adaptive BDI agents meet service-oriented programming. In P. Isaias and M. B. Nunes, editors, *Proceedings of the IADIS International Conference WWW/Internet 2005, Volume 2*, pages 205–209. IADIS Press, 2005.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, editors. *Extensible Markup Language (XML) 1.0 (Third Edition)*, 2004. W3C Recommendation 04 February 2004 downloadable from `http://www.w3.org/XML/`.

[8] A. Brogi, C. Canal, E. Pimentel, and A. Vallecillo. Formalizing web service choreographies. In *Proceedings of the First International Workshop on Web Ser-*

vices and Formal Methods, 2004. Published also in ENTCS, 105:73-94, 2004.

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

[10] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro. Choreography and orchestration: A synergic approach for system design. In B. Benatallah, F. Casati, and P. Traverso, editors, *Proceedings of the Third International Conference on Service-Oriented Computing (ICSOC 2005)*, pages 228–240. Springer-Verlag, 2005.

[11] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1, 2001. W3C Note 15 March 2001 downloadable from `http://www.w3.org/TR/wsdl`.

[12] P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering, 1st International Workshop, AOSE 2000, Revised Papers*, volume 1957 of *LNCS*. Springer-Verlag, 2001.

[13] J. Clark, editor. *XSL Transformations (XSLT) Version 1.0*. W3C Recommendation 16 November 1999 `http://www.w3.org/TR/xslt`.

[14] J. Clark and S. DeRose, editors. *XML Path Language (XPath) Version 1.0*, 1999. W3C Recommendation 16 November 1999 downloadable from `http://www.w3.org/TR/xpath`.

[15] Foundation for Intelligent Physical Agents. FIPA ACL message structure specification. Approved for standard, 2002-12-06. `http://www.fipa.org/specs/fipa00061/`, 2002.

[16] P. Giorgini, J. P. Müller, and J. Odell, editors. *Agent-Oriented Software Engineering IV, 4th International Workshop, AOSE 2003, Revised Papers*, volume 2935 of *LNCS*. Springer-Verlag, 2003.

[17] F. Giunchiglia, J. Odell, and G. Weiß, editors. *Agent-Oriented Software Engineering III, 3rd International Workshop, AOSE 2002, Revised Papers and Invited Contributions*, volume 2585 of *LNCS*. Springer-Verlag, 2003.

[18] H. Haas and A. Brown. Web Services Glossary – W3C Working Group Note 11 February 2004, 2004. `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#defs`.

[19] M.-P. Huget and J. Odell. Representing agent interaction protocols with agent UML. In *3rd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004)*, pages 1244–1245. IEEE Computer Society, 2004.

[20] M.-P. Huget, J. Odell, O. Haugen, M. Nodine, S. Cranefield, R. Levy, and L. Padgham. FIPA modeling: Interaction diagrams. First proposal, 2003-07-02. `http://www.auml.org/auml/documents/ID-03-07-02.pdf`, 2003.

[21] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, editors. *Web Services Choreography Description Language (WS-CDL) Version 1.0*. W3C Candidate Recommendation 9 November 2005 `http://www.w3.org/TR/ws-cdl-10/`.

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

[23] K. Mantell. From UML to BPEL – model driven architecture in a Web services world, 2005. Downloadable from `http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel/`.

[24] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an agent communication language. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Proceedings of the 2nd International Workshop on Agent Theories, Architectures, and Languages (ATAL'95)*, pages 347–360. Springer-Verlag, 1995. LNAI 1037.

[25] N. Mitra, M. Gudgin, and M. Hadley et al, editors. *Simple Object Access Protocol (SOAP) Version 1.2*, 2003. W3C Recommendation 24 June 2003 downloadable from `http://www.w3.org/TR/soap12`.

[26] Object Management Group. MOF 2.0/XMI mapping specification, version 2.1. Formal, 2005-09-01. `http://www.omg.org/docs/formal/05-09-01.pdf`.

[27] Object Management Group. Unified Modeling Language: Superstructure version 2.0. Formal, 2005-07-04. `http://www.omg.org/docs/formal/05-07-04.pdf`, 2005.

[28] J. Odell, P. Giorgini, and J. P. Müller, editors. *Agent-Oriented Software Engineering V, 5th International Workshop, AOSE 2004, Revised Selected Papers*, volume 3382 of *LNCS*. Springer-Verlag, 2004.

[29] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI–architecture. In *Proc. of KR'91*, pages 473–484, 1991.

[30] The OASIS Technical Committee. Universal Description, Discovery and Integration (UDDI) Version 3.0, 2005. Downloadable from `http://www.uddi.org/`.

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

[32] Wikipedia, the free encyclopedia. Web service, 2005. `http://en.wikipedia.org/wiki/Web_service`.

[33] M. Winikoff. Towards making Agent UML practical: A textual notation and a tool. In *Proc. of the 1st International Workshop on Integration of Software Engineeringand Agent Technology (ISEAT 2005)*, 2005.

[34] M. Wooldridge, G. Weiß, and P. Ciancarini, editors. *Agent-Oriented Software Engineering II, 2nd International Workshop, AOSE 2001, Revised Papers and Invited Contributions*, volume 2222 of *LNCS*. Springer/Verlag, 2002.

[35] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3), 2003.