Training compositional agents in negotiation protocols using ontologies

M. Amor, L. Fuentes* and J.M. Troya
Departamento de Lenguajes y Ciencias de la Computacion, University of Malaga, Campus de Teatinos s/n, E-29071 Málaga, Spain
E-mail: lff@lcc.uma.es

Abstract. The Internet facilitates the creation of new markets, which has motivated the development of new technologies for e-commerce. The increasing number of e-markets poses a big challenge to designers of e-commerce services. Agent-based applications are the most appropriate for merchandising because software agents are suitable for automating tedious tasks that the user may have to perform, such as, searching for goods. Furthermore, software agents are able to negotiate on the user’s behalf according to negotiation protocols. However, new challenges arise from the evolution of negotiation mechanisms. The lack of protocol interoperability at the application level is preventing the evolution of negotiation since the set of interaction protocols supported by a software agent is usually fixed once the agent is created, and any protocol adaptation needs agent replacement. Thus, new software technologies for the development of more flexible, adaptable and reusable software agents are needed. We propose the use of component technology for the development of adaptive software agents for new negotiations protocols. In this paper we describe the compositional architecture of a software agent, and how we take advantage of component orientation for training agents in new negotiation mechanisms at runtime.

1. Introduction

Nowadays the development of agent-based applications has increased due to the use of the Internet in daily life, especially with respect to e-commerce. Not only branded companies offer a web site for online shopping, individual users can also put their goods on sale on specialized sites, such as ebay [8]. In e-markets, software agents help users to negotiate, besides automating some commercial tasks. For instance, software agents are able to search among available offers, find the best suited to user preferences, and even negotiate in different e-markets, such as auctions.

Negotiation has long been recognized as an essential topic in multi-agent systems. The negotiation between two or more parties or software agents is carried out using a negotiation mechanism, also known as a market mechanism [17]. The negotiation mechanism consists of a negotiation protocol and a set of rules and constraints that determine the possible actions that participants can take. This mechanism must be public and shared by all software agents involved in the negotiation. The individual strategy adopted in the negotiation remains private, but it should be adapted to the negotiation protocol.

Emerging activity in this area [2,5,12,15,22] is currently focused on automated negotiation, which involves the design of high-level protocols for agent interaction. This research endeavours to make software agents more adaptive and flexible, capable of adopting new protocols that were not previously coded within the agents. However, most current multi-agent systems present important shortcomings related to negotiation protocol interoperability. Software agents participating in a negotiation mechanism must agree on the interaction protocol and must share the same assumptions about it. In most agent platforms [4,9,23], interaction protocols or more specifically negotiation mechanisms, are fixed in the agent implementation; in others words, they are hard-coded inside the agents. It would be desirable that e-commerce software agents

*Corresponding author.
could understand an open-ended list of interaction protocols, but this characteristic is not really supported by current software agents. This is an important limitation for agent interoperability in open markets.

The use of standards guarantees agreement about the negotiation protocol. FIPA [11] defines some standard interaction protocols for negotiation and different well-known auctions [10]. However, although FIPA interaction protocols can be parameterised for different protocol instantiation, they do not cover all the possible auctions or negotiation policies. Furthermore, these protocols only formalise interactions between software agents, and omit information related to other aspects and rules included in the negotiation mechanism [2]. Misplaced information could complicate agent participation in a negotiation (e.g. auction termination criteria could be different for two participants in an auction). Hence, a standard should either establish something more than the flow of messages between participants, including which negotiation rules have to be applied, or provide a standardized way of agreeing on these rules.

Another challenge in dynamic negotiation is facing the evolution of a negotiation mechanism. If the specification of a negotiation protocol changes, all the software agents that comply with it should be updated. Thus, the cost of producing new agent versions depends on the adaptability and extensibility of the software agent model. Unfortunately, these features related to software evolution are not widely taken into account in the development of software agents.

It is commonly agreed that component-based software engineering (CBSE) provides advanced techniques that can be applied to the design and development of software agents [3,20] by plugging reusable software components. We propose a component-based architecture for developing software agents that deals with some of the challenges explained above [1]. The benefits of component-oriented systems stem from their increased flexibility: a system built from components should be easier to reconfigure to address new requirements [18]. A compositional design may allow software agents to be dynamically adapted to support new negotiation protocols and new functionality. Our goal is to endow software agents with a high-level mechanism for their runtime adaptation, enabling them to participate in any negotiation regardless of the interaction protocols supported.

Agent behaviour is provided by different plug-in components. In order to improve agent modularisation we apply the separation of concerns principle [16]. Concretely, functionality and coordination issues are separated internally into different entities. The composition between agent internal components is performed at runtime, allowing the reconfiguration and adaptation of agent behaviour to support new interactions.

To evaluate our contribution, the model is applied to the e-commerce domain. One of the most extended negotiation mechanisms in e-commerce is the auction. The Internet hosts many auction sites that work like traditional auction houses, where software agents are used to automate the negotiation. A key aspect of these systems is the richness and flexibility of inter-agent interactions.

The benefits of our proposal can be successfully applied in “real” scenarios of open markets, where software agents should be more flexible and interoperable. For example, suppose a user called “Peter”, who is interested in buying the first edition of Shakespeare’s famous novel “Othello”. After searching for it in several bookstores, he decides to try on the Internet, and he launches a software agent to bid on his behalf. But the only place that offers a special edition of the desired book is an auction site where the type of auction is the Vickrey auction, and Peter’s agent only supports the English auction. Well, if Peter decided to use one of our software agents, he could have a last chance. Peter’s agent can be trained to participate in this auction following Vickrey auction rules. Peter’s agent will be trained to understand the new protocol without changing its internal functionality or losing Peter’s preferences (desired product, maximum price, etc.).

2. Software agent compositional model

In this section we describe a compositional model for software agents. Software agents are systems that perform actions or tasks autonomously to reach a set of goals. Our model encapsulates agent functionality in software components. Modelling agent functionality as components, we find that a software agent contains components for storing data (e.g. to store its internal goals, facts, etc.), or for providing application domain functionality (e.g. buy, bid, search), and components that coordinate the agent internal and external execution.

In order to improve agent functional decomposition, we apply the principle of separation of concerns, that is, we model the coordination issue separately in an entity called connector. By coordination we mean the flow control between agent-inner components and/or external agents. Thus connectors, for instance, model
negotiation protocols. In a previous work we successfully applied this paradigm [13,14] in component-based Web applications. In this case, the separation of coordination aspects facilitates the dynamic adaptation of agents to new interaction protocols. This is performed by the runtime composition of components and connectors inside the software agent.

Figure 1 shows the UML class diagram of the proposed compositional architecture of a software agent based on components and connectors. We use UML stereotypes for modelling the principal entities of our model, which are Component, Connector, Mediator and Interface.

Connectors coordinate the different interactions or conversations in which the agent is involved. An agent can participate in more than one interaction or conversation simultaneously, so a protocol connector controls each conversation. Connectors coordinate dialogues according to a specific protocol (e.g. English Auction Negotiation Protocol). These connectors differ only in the coordinated protocol, given by a template (Protocol Template class in Fig. 1) filled by the protocol specific rules, transitions and constraints. Thus, an agent is able to encompass a new protocol only by downloading the corresponding template, as we show in the following sections.

Components encapsulate data and behaviour. Some behavioural components are always present in the architecture providing the generic agent functionality (Basic Agent Actions and TpfN Actions components), such as sending a message, or storing the general data (KnowledgeBase component). Domain specific behaviour is also provided by components (e.g. e-market component). These components may be plugged into agent functionality on demand, and are easily changeable over the lifetime of the agent, guaranteeing software agent evolution. In Fig. 1, the e-market component offers functionality to perform e-commerce tasks, such as generating a bid or making an on-line payment.

Interface components manage agent’s external interactions. They process the input messages and send replays to an agent platform for delivery. The component Agent External Communication (AEC) encapsulates the external communication interface of the agent. This interface component deals with incoming and outgoing messages. Incoming messages are first processed by this component, and it may discard messages which are not syntactically well-formed. The component Agent Interface (AI) contains the public agent interface, which is an extension of traditional IDLs (Interface Definition Language) of software components. For instance, AI includes, in addition to the description of the provided services, the public negotiation protocols, the templates of the supported protocols, ... (providing a detailed description of the agent interface is beyond the scope of this paper).

The mediator component Agent Compositional Core (ACC) mainly performs the dynamic composition of components and connectors. The ACC receives input messages from the AEC component, dispatching them to the appropriate conversation according to the conversation identifier contained in the message. In addition, this component launches a connector encapsulating a specific protocol as a new interaction is initiated. Dynamic composition is used for the late binding between coordination connectors and the agent behaviour provided by components. Regarding evolution issues a relevant characteristic of our model is that components and connectors have no direct references among them, decreasing their dependency. For instance, the ACC will link an auction connector to the e-market action component, but this will be done at runtime. Dynamic composition also allows plugging a new component implementation into the agent dynamically, or upgrading this compositional agent without replacing it.

The next section shows the execution of a protocol through an example. Concretely, we will describe the protocol that instructs the agent in a new negotiation protocol.

3. Agent protocol training using ontologies

In this section we will show the training protocol of a new negotiation protocol. By training we mean telling somebody how to do something. In our case somebody is the agent, and the action is how to follow a protocol. Above, we showed that a software agent has enough functionality to use a negotiation mechanism since it can send messages, receive messages, and do a particular activity (sell, buy, . . .). Imagine that Peter’s agent is interested in participating in an auction of books, but it does not understand the negotiation protocol. In our proposal we overcome this problem by training Peter’s agent in the new negotiation protocol. Then, the auctioneer agent will train Peter’s agent through the Training Protocol for Negotiation (TPfN), which must be accepted by both agents. In our approach TPfN is the only protocol that must be known by all agents. During execution of this protocol, the trainee agent receives a protocol description (the template commented above) that contains the protocol state
and they are also used in protocol descriptions. Since services are described as part of the agent interface, they can be invoked during a protocol execution. Agent public is necessary to describe the agent services that may be required during a protocol execution. As mentioned in the previous section, it can be taken as a coordinated collection of agent services that should share. Then, we are going to describe how negotiation protocols are represented in order to be understandable for the trainee agent. Finally, we will show how the TPIN protocol, in others words the training machine (STD) with the actions that the agent has to perform during a negotiation, for example an auction. The training protocol does not add functionality to the agent, but increases its interaction capability. In this section we are first going to present an ontology for describing agent services, which is used to define the common knowledge that the trainer and trainer agents should share. Then, we are going to describe how negotiation protocols are represented in order to be understandable for the trainee agent. Finally, we will show how the TPIN protocol, in others words the training protocol, works.

3.1. An ontology for describing agent services

An agent provides a set of functionalities that can be considered as services it offers to the environment. The participation of an agent in an interaction protocol can be taken as a coordinated collection of agent services invocations. As mentioned in the previous section, it is necessary to describe the agent services that may be invoked during a protocol execution. Agent public services are described as part of the agent interface, and they are also used in protocol descriptions. Since our goal is to train agents in a new negotiation protocol, and a protocol specification includes references to agent-internal actions, both the trainer and the trainee should use the same terms and semantics to refer to agent actions. So we propose the use of ontologies for describing the actions that an agent can perform and that may be required during a protocol execution. Regarding our compositional agent architecture described in Fig. 1, these actions are contained in components.

We propose to use DAML-S [6] to describe the agent provided and required functionality. DAML-S is a DAML+OIL [7] ontology for describing the properties and capabilities of Web services, and it is part of DARPA Agent Markup Language program. DAML-S provides Web services descriptions at the application layer, describing what a service can do, and not how it does it. DAML-S is a Web markup language that provides a declarative, computer-interpretable API for executing function calls, which is exactly what we need. Since DAML-S is more oriented to Web services, we adapted it to our particular requirements.

We will take a software component for e-commerce as an example of how to define and describe a software component public interface using DAML-S. This soft-
ware component defines a collection of services (e.g., bid, buy) accessible through its public interface. The description of a service using DAML-S comprises three different views of the service: what the service requires and provides the user or agent is described in the service profile; how the service works is given in the process model; and how to use the service is described in the grounding.

The first step in describing a software component public interface in DAML-S is to declare each individual method in the interface as a service. One instance of Service will exist for each distinct published service of the software component IDL (Fig. 2 shows the definition of the Bid service).

The Service class is at the top of the DAML-S ontology. Service properties at this level are very general: the property presents refers to the profile that describes the service capabilities; the property described By refers to the service model that describes how the service works; and the property supports refers to the description of how to invoke the service.

A service profile provides a high level description of a service and its provider. It is used to request or advertise services for service discovery and registry purposes. So the next step is to describe the profile of the Bid service, given in Fig. 3.

Service profiles consist of three types of information: a textual description of the service and the service provider (in our case, the software component), such as the service name and some contact information; the functional behavior of the service; and several functional attributes for automated service selection (for example, the quality rating). Going back to the example, Fig. 3 shows the profile (referenced as Bidding-Service) of the Bid service, including input and output parameters (InitialAmount and BiddedAmount). Note that service providers use the service profile for advertising their services, but service requesters also use the profile to specify their needs and expectations. For instance, a provider might advertise a software component that provides a service for participating in an auction, whereas an agent may look for a software component that reports current auction houses catalogs. DAML-S service profiles focus on the representation of what the service does rather than where to find the service. Since in open environments, such as the Internet, one is not assured of having resources available at any given time, where to find a service as part of the service description can even represent a problem. Therefore, the use of DAML-S can be considered an improvement for finding a service. In addition, DAML-S service profile description allows to determine all services that provide the same functionality and that are available at the moment of the service search.

The third step is to provide a declarative description of the service properties in a process model. At present, the process model describes the individual programs that comprise the service, and conceives each program as either an atomic, simple or composite process. Since we are describing a software component interface, each service will map into an atomic process. In DAML-S, a non-decomposable program is described as an atomic process characterized by its ability to be executed by a single call, like a method in a software component interface. Thus, the Bid service is described in the process model as an atomic process, as shown in Fig. 4.

A set of properties describing the service parameters classified as input and output is associated with each process. An example of an input of the Bid service is the last amount offered by customers, declared in Fig. 4 as the property inputBid. Inputs can be mandatory or optional. In contrast, outputs can be described as conditional. For example, when you search for a product in a catalog, the output may be a detailed description or an “unclassified product” response. In addition, inside the process model it is possible to describe preconditions and conditional effect properties of the service, as well as inputs and outputs parameters.

Finally, for each atomic process it is necessary to describe the grounding. The grounding of a service describes how to use the service in a concrete realization. For a web service, the grounding description consists of the atomic processes described in the process model and the concrete SOAP (Simple Object Access Protocol) messages that are sent to access the service. Hence, the grounding details about service access can be considered as a mapping from the general description given in the process model to a concrete implementation of services, for instance in a software component platform. The main function of the grounding is to provide the description of the access method that can be used for accessing the Bid process (we have declared that the Bid service is realized by the atomic process Bid). For instance, method invocation will be used in case the software component is a Java class, or message passing for a Web service-oriented implementation.

If an agent commits the e-market ontology, this means that there is an internal component that implements the ontology functionality required (in this case the service Bid with its corresponding inputs and outputs). According to object-oriented terminology, this component is a container of the methods defined as ser-
services in an ontology, or, at least, it has defined mappings between its functionality and the actions defined in the ontology as services. In addition, inside the agent architecture this component is denoted inside connectors by the identifier of the ontology it commits, instead of using direct references. As components and connectors are not coupled entities, we can change components implementations at runtime. This is a consequence of separating component definition from implementation details. The mediator component that performs the dynamic composition will resolve which component provides the required service according to a given ontology. In the example, an auction connector will invoke the bid action defined in the e-market ontology, so the mediator will redirect this invocation to a concrete implementation (the current one) of the e-market component.

3.2. Description of negotiation protocols

In this section we are going to describe how we specify the negotiation protocols that a trainer agent will send to any agent that wants to be trained in that protocol. The specification of the interaction mechanism is not a trivial issue, as there are many aspects to consider. A simple description of the flow of interchanged messages is not enough, because it is also necessary to express the actions that the agent performs during
interaction. When an agent is instructed in a new interaction mechanism, it needs to know the relationship between the protocol input and output messages and its internal behaviour. Now we are going to describe how we specify a template for a negotiation protocol. The negotiation mechanism is described using XML [26], a standard language for representing structured information and the basis for Web mark-up languages like DAML-S. The essence of XML is the separation of abstract content from presentation. A major advantage of using XML is the range of XML tools and APIs that can be used to interpret this language. In addition, XML is programming language-neutral.

Now, we present how to describe a negotiation mechanism in XML through an example. The example is based on the Vickrey auction protocol, which is not a FIPA standard but is FIPA compliant, and can thus be implemented in any agent platform. The corresponding state transition diagram (STD) is given in Fig. 5.

In the diagram, every transition is labelled with the input message that triggers the transition, and the output message that is sent if fitting. As can be seen, it is a very simple protocol. In a Vickrey Auction, also known as second-price sealed bid, the bids are submitted to the auctioneer without knowing the bids of the others. The auction participant that offers the best bid wins the auction, but pays the second highest amount bid [19].

Now we will describe the overall structure of the negotiation protocol description, which we identify with a ProtocolID (Fig. 6). The description also specifies the ontology of services that is referenced along the protocol description which in the example is the e-market ontology identified as OntologyID. The XML schema in Fig. 6 shows that the protocol is mainly described in terms of messages (either input or output messages), states, transitions, and state transition rules.

Some of these concepts are described again using DAML-S because it provides standard descriptions for data types, processes, parameters, control constructors, and data binding for relating the inputs and outputs of process composition.

Firstly, the specification describes every message that may be interchanged between the protocol parties. The description of a message has to include the informative and should contain at least a description of the content. An identifier can be assigned to the message description to later refer to it in the specification, inside the rules and transitions description. Figure 7 shows the description of the cfp (Call for proposal) message of a given protocol and an example of a message sent by the auctioneer advising Peter’s agent that a book is going to be auctioned. In the example we use XML as content language. The XML message description in Fig. 7 shows a message identified as cfp and whose content includes the auctioned product (product element). The description of the content structure and other properties is done in DAML-S (see again Fig. 7).

The second section of the protocol description specifies the initial state and the following ones. The list of states of the Vickrey auction protocol is given in Fig. 8.

The third section describes the transitions, specifying each transition as a set of actions. The execution of a transition implies dynamic method invocations on internal agent components. Each transition is described using a DAML-S class Process. Like DAML-S processes, we can express atomic, simple or composite transitions. Usually, a transition is going to be declared as a composite process (1 in Fig. 9), and we also describe the set of actions that have to be invoked, and how they are composed by using a control constructor. These actions are composed by using control constructors. A composite process must have a composeOf property that indicates the control structure of the agent composite actions (2 in Fig. 9). DAML-S defines the following control constructors: i) Sequence, for executing a list of processes in a sequential order; ii) Concurrent, for executing elements of a bag of processes concurrently; iii) Split, that invokes elements of a bag of processes; iv) Split+Join, that invokes a bag of processes and synchronizes; v) Unordered, to execute all processes in a bag in any order; vi) Choice, to choose between alternatives and execute one; vii) If-Then-Else, if the specified condition is true, it executes Then, otherwise it executes Else; viii) Repeat-Until, that iterates the execution of a bag of processes until a condition is true; ix) Repeat-while, that iterates the execution of a bag of processes while a condition is true. In the
example, the actions are composed using the sequence control constructor (3 in Fig. 9).

Each control constructor is associated with a property called components to indicate the list of processes that are composed. In the example, the transition consists of the sequential execution of the Bid and the SendMessage services (4 in Fig. 9). The process model of these processes is described in their corresponding ontologies. The Bid action is defined inside the e-market component, and the SendMessage belongs to the Basic Agents Actions component (Fig. 1).

To define composite process using DAML-S, it is necessary to relate the inputs and outputs of its component subprocesses. In a programming language, these elements are related using variables. But DAML+OIL does not support variables. In DAML-S, the way to re-
late two properties to make their values equal is by using the property sameValues. Figure 10 shows how to describe that the output parameter outputBid of the process Bid is bound to the parameter contentField, which refers to an input parameter of the process SendMessage.

Finally the state transition rules that condition the interaction mechanism are described. These rules specify, for a given state and an entry message, which transition is executed and which state is the next one. In addition, the application of a rule can be conditioned by the evaluation of an action. Figure 11 shows the description of one rule of the Vickrey auction protocol. This rule is applied when the agent is waiting for the auction to start (state StartAuction) and the cfp message arrives. This rule is only conditioned by the current state (element StateIdent is StartAuction) and the input message (cfp message), both described by their identifier. In addition, the triggered transition and the next state are described by their identifier (transition 1 and WaitingForInform).

3.3. The TPfN protocol

As we said in the previous section, the training is performed through the coordination protocol TPfN. A
message sequence diagram of TPfN is given in Fig. 12. The participants involved in the training process are the trainer and the trainee agents. The trainer agent sends the trainee agent the description of a new negotiation mechanism.

3.3.1. TPfN protocol description

This protocol is composed of two FIPA interaction protocols. The interaction starts when the trainer queries the trainee agent about whether it accepts or not to be trained in a new negotiation protocol (The Query Interaction Protocol [10] in Fig. 12). This decision is made by the agent based on the minimum required functionality suggested by the trainer in the query message, and its own internal constraints. The trainee agent notifies whether it accepts to be trained with an inform message or not with a refuse message. If the trainee agent wants to be trained, then the training itself is initiated (Request Interaction Protocol [10] in Fig. 12). The training process consists in sending a negotiation protocol description in the content part of a request message. In the example, the trainer agent, for example the auctioneer agent, asks the bidder agent whether or not it understands the Vickrey auction protocol with the semantics defined by the e-market ontology.

TPfN protocol is defined using FIPA communicative acts and protocols, in such a manner that can be implemented by any FIPA compliant agent platform, using XML as the message content language.

Internally, training is performed through a protocol. It is thus integrated in the agent architecture by a protocol connector and a component that commits the TPfN ontology (TPfN Actions component of Fig. 1). The TPfN ontology defines three services that are invoked during training, namely UnderstandProtocol, VerifyProtocol, and TrainProtocol.

On the trainee agent side, the training process consists in translating the negotiation protocol description to a template. A negotiation protocol template is represented internally by the protocol state transitions rules and the description of the transitions as a set of agent actions (see again the ProtocolConnector entity of Fig. 1).

TPfN protocol transitions execute actions included in the TPfN Actions component and in components that provide basic agent behaviours. While VerifyProtocol and TrainProtocol actions refer to TPfN specific actions, SendMessage, which is invoked when the agent sends a message to another agent, is part of the agent basic functionality provided by the component Basic-AgentActions in Fig. 1.

The TPfN protocol execution is as follows: when the first message of TPfN protocol is received (query-if message in Fig. 12), the Mediator component creates a connector parameterised by the TPfN template. The implementation of the protocol connector is the same for all protocols; thus, connectors only differ in the protocol template provided on instantiation. Once the TPfN connector is created and initialised (the current state is set to the initial state given in the template), the Mediator component forwards the first message (Query-if) to the TPfN connector. The connector checks which state transition rules are applicable or active. A rule becomes active when the state and the message conditions are fulfilled. Afterwards, the TPfN

\[
\begin{array}{llll}
\text{rules} & \{\text{id}\} & \{\text{stateId}\} & \{\text{messageId}\} & \{\text{transitionId}\} & \{\text{nextStateId}\} \\
1 & \text{Rule1} & \text{StarAuction} & \text{cfp} & \text{Transition1} & \text{WaitingForInform} \\
2 & \text{Rule2} & \text{WaitingForInform} & \text{inform} & \text{Transition2} & \text{EndAuction} \\
3 & \text{Rule3} & \text{StarAuction} & \text{cfp} & \text{Transition3} & \text{EndAuction}
\end{array}
\]

Fig. 11. XML rule description.
connector chooses only one, executes the corresponding transition, and modifies the current state according to the rule.

In the example, only one rule is successful and the connector invokes the action UnderstandProtocol(protocol-name, ontologies). As mentioned before, this action is performed by TPN Actions, which decides if the agent accepts to be trained, based on the new protocol and the ontology specified in the query-if message content.

Protocol execution continues if the agent accepts to be trained. An inform message is sent, and the connector waits for the new protocol description to come. When the message containing the XML protocol specification arrives, the TPN connector performs the new protocol validation and training from the specification received. For that, the TPN connector invokes the VerifyProtocol action again over the TPN Actions component. This component uses the introspection\(^1\) property to check whether the agent commits the ontology referred to in the specification (if the agent has the corresponding components that implements the said ontology); and to check if the agent supports the protocol message content language (if the agent has an appropri-

\(^1\text{Introspection is the ability of a component to say something about its behavior. Component interfaces give information about the services offered, services required, quality of service, how to customize the appearance and behavior of a component and so on.}\)
ate language parser); moreover, the component could perform a formal verification, which is optional.

If the protocol specification was successfully validated, an agree message is sent. The next step is to translate the XML description to a new protocol template, and add the new protocol to the agent interface. This is done by the TrainProtocol action of the TPfN Actions component. This action generates the protocol template, and updates the AgentInterface component by adding the new template. After that, the protocol execution ends.

3.3.2. The execution of a new negotiation protocol

Now we are going to briefly explain the sequence of interactions among components inside the agent when a new conversation is initiated (see Fig. 13). Let us take an agent that receives the first message according to the Vickrey protocol. As shown in Fig. 13, the software agent receives the message through the AEC component, which has the capability of sending and receiving any kind of message through any transport protocol or component platform. If the message is syntactically well formed, it is forwarded to the Mediator component (1 in Fig. 13). This component becomes aware that this is the first message of the protocol, because the conversation identifier contained in the message is new. Consequently, the Mediator has to instantiate a protocol connector that will control the new interaction based on the Vickrey protocol (the protocol identifier VickreyAuction is contained in the message). The protocol template of the Vickrey protocol is retrieved from the AgentInterface component (2 and 3 in Fig. 13). The Mediator component then instantiates a protocol connector, and once it is created, every time a message belonging to that conversation arrives, it is notified of this circumstance. For example, the message labelled 5 in Fig. 13 is used to notify the protocol connector that a cfp message has arrived and has to be processed. The protocol follows the rules described in the example in last section. When the agent receives a call-for-proposal message, the protocol connector will invoke the action to generate a bid (action e-market, bid, “100”), see an example in Fig. 13. The Vickrey protocol connector does not directly invoke the action, but it forwards the request to the Mediator component (6 in Fig. 13).

The Mediator component will forward the service request to the component that provides this service and will send the result back to the connector. So in some way, this component works as a service broker. Notice that the Mediator performs the dynamic composition of components and connectors by handling the connector’s output calls, and delegating them to the appropriate software component. Going back to our example, since the Mediator has the reference of the e-market component, it maps the general request “action” (message 6 of Fig. 13) to a concrete invocation, after consulting the grounding for the selected component. In this case, the Mediator component performs a method invocation using a load-time reflection mechanism (message 8 in Fig. 13). Finally, the Mediator will receive the result of the operation (9 in Fig. 13), and will pass it to the connector that continues the protocol execution (10 in Fig. 13).

4. Related work

Research in automated negotiation focuses on the design of protocols and associated strategies. Automated negotiation relies on the idea that agents have to use a shared protocol for resolving issues that can occur in the negotiation. However most multi-agent systems are designed with negotiation protocols explicitly hard-coded inside the agents that participate in a negotiation. Two important limitations come up as a consequence of this restricted design: It is not possible to extend or update the protocol specifications without replacing the agent; and only those agents that were designed to use a specific protocol are accepted for a negotiation. Furthermore, the coordination protocol and the related agent internal behavior are tightly coupled, which makes it difficult to manage the compatibility, evolution and reusability of software agents. Hence, in order to be used in open environments like the Internet, software agents should be allowed to choose the most suitable interaction in which to participate. For this reason, one of the challenges of agent design is to endow software agents with flexible mechanisms to support almost any interaction protocol.

With respect to this challenge, some recent alternative proposals have been formulated. Although all of them have characteristics in common with our work, they try to approach it from different perspectives. The approach to automate negotiation proposed in [22] is based on the idea that the negotiation protocol should not be hard-coded within the agent, but that it should be acquired from the marketplace where the negotiation takes place. The protocol is described in a DAML+OIL ontology, and it consists of a set of declarative rules that describe the negotiation. When the agent participates in a negotiation, it has to interpret the DAML+OIL ontology, and translates it to a set of declarative rules,
which are fed into a JESS rule engine. The proposal is comparable to ours, since ontologies are used to achieve an agreement about the terms used in the negotiation protocols. Like in our approach, the negotiation protocols do not need to be previously coded as part of the agent, but can be learned dynamically by acquiring the ontology. However, in this approach agent behaviour is part of the protocol description, expressed in terms of declarative rules as we explained above. In contrast, our proposal separates protocol rules from agent behaviour, which facilitates the (re)use of agent functionality implementations across different negotiation protocols. For instance, this approach prevents agents implementing different kinds of auctions of (re)using the bidding code. Besides, this approach, which is only applied to auction protocols in their early stages, is useful mainly when the negotiation can be reduced to a set of declarative rules that can be interpreted by a rule engine. The proposal in [2] relies on the same bases, but it does not provide an ontology for describing negotiation protocols. Instead, the authors propose a taxonomy (that can be considered an ontology) of declarative rules, which can be used to describe a wide variety of negotiation mechanisms. In addition, the negotiation communication is performed by a simple interaction protocol that supports any negotiation mechanism described using these declarative rules. They also implement in JADE a software framework that allows agents to participate in negotiations defined using their rule taxonomy and interaction protocol. Although this approach also appears to be similar to ours, it is not clear whether they can describe any negotiation protocol. They provide a limited set of declarative rules, which could constrain the kind of negotiation protocol that can be described. In our approach this drawback is overcome using an extensible description that is not restricted either to a fix set of rules or to specific negotiation protocols.

A different approach to provide flexibility to software agents is to upload the external code that is executed by the software agent as it becomes part of the agent functionality. With respect to this viewpoint, the approach presented in [5] supports a dynamic behaviour modification of agents. This kind of dynamic agent is not designed to have a fixed set of functions; instead, agents are able to carry out application-specific actions written
We use ontologies to separate the implementation from the actions, reducing the reusability and evolution of both agent behaviour and coordination. However, although we take advantage of a similar concept – the behaviour of our agents is provided by plug-in components – the software components plugged into our agents do not enclose any coordination information, so they can be reused and updated without worrying about the interaction protocols in which they are involved. In addition to this important drawback, this framework does not seem to be able to interoperate with other agent systems, and the uploading of external codes at runtime is not efficient, with the possibility of posing significant security problems. In the work presented in [15], agent behaviour is also provided by an external code. The authors define protocols as policy packages, which are exchanged and installed in agents at runtime. A policy package is a XML document that describes the protocol for each agent role, which consists of a set of state-transition rules and transitions. When a transition is performed, an external Java class method is invoked. This proposal is closer to our approach than the previous one. Like our proposal, it separates coordination, defined by state-transition rules, from agent behaviour, provided by Java classes. The Java code is executed when the rule that conditions the transition is applied. By defining different policy packages, this separation should allow for reusing the same behaviour across different protocols and the same protocol rules by different transition programs. However, agent functionality cannot be easily reused since the message delivery or the invocation of other internal Java methods are implemented for a specific behaviour inside a protocol. We use ontologies to separate the implementation from the description of the provided service. Within the protocol description, agent behaviour is referred to by the corresponding ontology, instead of using a direct reference to the component that implements it, like in the previous approach. Thus, we can reuse a software component across different protocols if it commits the ontology description required by the protocol specification. Another significant shortcoming of the last approach is that the agent does not have control over the external code. In our approach, software components providing new functionality are plugged once at the beginning of the interaction, but not every time their functionality is needed. We try to increase reusability of behaviour using independent software components providing fine-grained services that can be composed to provide a more complex functionality. This composition is specified in the protocol representation using the wide variety of control constructors provided by DAML-S.

Another very recent approach that explicitly represents protocols in XML is given in [12], where the authors propose a new mechanism to execute these protocols. The XML protocol representation is based on the authors’ AUML protocol description. AUML is a software engineering methodology that extends UML interaction diagrams to handle agent interaction protocols. Although this notation is useful, it only considers the sequence of message exchange. Therefore, AUML does not encompass the complete content of the exchanged messages, does not provide information about the decisions that should be made during the execution, and does not specify how to bind input and output messages with the agent internal data and behaviour. The main contribution of this approach is the dynamic generation of an executable code that causes the agent to comply with the desired protocol. This executable code consists of production rules implemented in JESS. Consequently, the authors translate an incomplete protocol representation to a set of production rules that provide the agent behaviour and control the agent interaction. In comparison with our proposal, this approach is much simpler. Our XML protocol description presented in previous sections includes all the information defining a complete protocol representation, modelling not only the state-transition diagram that describes the exchange of messages, but also the detailed description of each message content format, and a more complex agent behaviour using the functionality provided by software components.

We want to point out that our proposal has some points in common with some of these works, but it goes a bit further, overcoming the different shortcomings exposed above.

5. Conclusions

In this paper, we propose a component-based architecture for software agents that combines component
orientation and separation of concerns. Agent domain specific functionality and coordination are modelled as separate entities. Functional components (e.g. e-commerce) are bound at runtime and linked with the corresponding negotiation protocol connector.

The flexibility provided by component orientation enables software agents to support new negotiation protocols. We outline the TPIN protocol for runtime training a software agent in a new negotiation protocol specified in a XML schema. This mechanism increases agent interoperability regarding protocol interaction support inside an application domain. We have illustrated the benefits for the e-market domain, where an agent is not forced to negotiate through a fixed set of negotiation protocols.

In general, the use of ontologies ensures that the agents agree on the meaning of some terms. An ontology can be explicitly represented, or it can be implicitly encoded within the actual software implementation of the agent without being formally published. It is clear that the former approach demands more flexible software agents that will be able to commit ontologies explicitly represented. An ontology should be extensible to adapt to new requirements inside an application domain. On one hand, an ontology should be general enough to cope with the needs of several application domains, but, on the other, it also requires some specialization to handle knowledge concerning a particular domain. With respect to these features we decided to use the DAML-S ontology for describing the software component public interfaces used in protocol descriptions, and also to take advantage of how DAML-S addresses the representation of process composition to compose agent internal behaviour. The service representation of DAML-S is much richer than the representation provided by emerging standards such as Microsoft’s UDDI [24], or W3C’s WSDL [25], because DAML-S enables including any semantic description. Another feature that led us to use DAML-S (and also DAML+OIL) is that using a standard technology facilitates the acceptance and reusability of the proposed ontology. Our approach is only limited by the current state of this technology, which imposes compliance with the latest release. All the processing of a XML protocol description and software component ontologies is performed using Java APIs JAXB (Java Architecture for XML Binding) and JAXP (Java API for XML Processing) [21], which create Java classes from the XML schema and the DAML-S ontology defined in previous sections.

We are now improving a FIPA compliant working prototype of a generic software agent using the compositional architecture outlined in the previous sections.

References