Frontiers in Artificial Intelligence and Applications

FAIA covers all aspects of theoretical and applied artificial intelligence research in the form of monographs, doctoral dissertations, textbooks, handbooks and proceedings volumes. The FAIA series contains several sub-series, including "Information Modelling and Knowledge Bases" and "Knowledge-Based Intelligent Engineering Systems". It also includes the biennial ECAI, the European Conference on Artificial Intelligence, proceedings volumes, and other ECCAI – the European Coordinating Committee on Artificial Intelligence – sponsored publications. An editorial panel of internationally well-known scholars is appointed to provide a high quality selection.

Series Editors:

Volume 220

Recently published in this series

Vol 218  C. Soeters and R. Ghazi (Eds.), Data Mining for Business Applications
Vol 216  P. Baroni, F. Ceriotti, M. Giacomini and G. B. Sonnari (Eds.), Computational Models of Argument – Proceedings of COMMA 2010
Vol 215  R. Coelho, R. Studer and M. Wooldridge (Eds.), ECAI 2010 – 19th European Conference on Artificial Intelligence
Vol 213  L. Olms, I. Janssen and W. Custers (Eds.), Ontologies and Semantic Technologies for Intelligence
Vol 212  A. Respicio et al. (Eds.), Bridging the Socio-Technical Gap in Decision Support Systems – Challenges for the Next Decade
Vol 211  J. du Silva Filho, O. Lambert-Torres and J. M. Abe, Uncertainty Treatment Using Paraconsistent Logic – Introducing Paraconsistent Artificial Neural Networks
Vol 210  O. Kutz et al. (Eds.), Modular Ontologies – Proceedings of the Fourth International Workshop (WoMO 2010)
Vol 209  A. Galton and R. Mizoguchi (Eds.), Formal Ontology in Information Systems – Proceedings of the Sixth International Conference (FOIS 2010)

IOS Press

Amsterdam • Berlin • Tokyo • Washington, DC
Contents

Preface  v

René Alquézar, Antonio Moreno and Josep Aguilar

Conference Organization  vii

Invited Talks

Towards Language-Competent Web Search
Veronica Dahl  3

Autonomy and Artificial Intelligence in Aerospace Systems
Juan de Dalmau  5

Agents and Multi-Agents Systems

Norms Evaluation through Reputation Mechanisms for BDI Agents
Isaac Payol, Roberto Centeno, Ramon Hermoso,
Viviane Torres da Silva and Jordi Sabater-Mir  9

Cultural Algorithms: Less is More
Esteve Abreu and Ulises Cortés  19

An Agent Architecture for Simultaneous Bilateral Negotiations
Angela Fabregues and Carles Sierra  29

Simple Ecological Rules Yield Complex Agent Networks
Miguel Lurig and David Robertson  39

Comparison of Topologies in Peer-to-Peer Data Sharing Networks
Jordi Camps, Núria Piquel, Maití Lopez-Sanchez
and Marc Esteva  49

Argumentation-Based Example Interchange for Multiagent Induction
Santiago Ortún and Enric Plaza  59

COnCENs: Organizational Awareness in Real-Time Strategy Games
Sergio Alvarez-Napagao, Ignasi Gómez-Sebastià,
Javier Vázquez-Salceda and Fernando Koch  69

A Flexible Agent-Oriented Solution to Model Organisational
and Normative Requirements in Assistive Technologies
Ignasi Gómez-Sebastià, Dario Garcia-Gasulla, Cristian Barué,
Javier Vázquez-Salceda and Ulises Cortés  79

Towards Dynamic Agent Interaction Support in Open Multiagent Systems
Ricard L. Ponsà, Juan M. Alberola, Jose M. Such, Agustín Espinosa
and Ana García-Fornes  89
Towards Dynamic Agent Interaction Support in Open Multiagent Systems

Ricard L. Fogués, Juan M. Alberola, Jose M. Such, Agustin Espinosa, and Ana Garcia-Fornes

Departament de Sistemes Informàtics i Computació
Universitat Politècnica de València

Abstract. Open Multiagent Systems, in which heterogeneous agents interact with each other and organize themselves into Virtual Organizations, demand infrastructures supporting these features. In these systems, dynamic and complex interactions between agents may arise. Interaction Protocols allow the definition of communication patterns. However in open systems, dynamic and complex interactions may also require these patterns be modified at execution time. We propose a support for modelling complex, concurrent and dynamic interactions between agents in terms of conversations. Conversations between agents follow predefined Interaction Protocols that can be dynamically modified without restarting the system. This support is provided at agent level and is integrated into the Magentix Multiagent Platform.

Keywords. Dynamic Interaction Protocols, Agent Conversations

1. Introduction

As stated [16], Multiagent Systems (MAS) allow the development of complex computational systems in open environments. Agent-based technologies are involved in a wide range of domains, specially identified as open and dynamic environments in which agents can act on behalf of service owners, locating services, negotiating contracts and making pro-active runtime decisions while responding to changing circumstances. This view of systems demands the development and integration of infrastructures in which agents with different capabilities are able to collaborate and create coalitions by means of Virtual Organizations.

Practical applications based on MAS technology will be mainly realised in the future in terms of open systems (Electronic Marketplaces, Virtual Enterprises, Virtual Organizations, etc.) where members are developed by different parties and have conflicting goals [9]. In the last few years, many researchers have focused their efforts on these kinds of systems: electronic institutions [11], Virtual Organizations [6], etc.

As open systems are very changing environments, they must be able to dynamically adapt their structure and behaviour by means of adding, removing or substituting components of the system while it is running and without bringing it down [20] [10]. This

---

1 rlopez@dsc.upv.es
2 Universitat Politècnica de València. Camí de Vera s/n. 46022, València
dynamic adaptation could also involve social issues relevant to the agents, that is, how agent interactions are carried out.

In these open systems complex interactions between agents can change dynamically. IPs (IPs) allow the specification of the agents’ behaviour within a closes environment in which allowed interactions are predefined. However in dynamic scenarios interactions can be modified at execution time in order to adapt the system to new requirements: a new norm can appear in order to book a hotel room, an interaction with a loan service may require that additional information be requested because several clients have not been honest, an auction protocol could be modified because many clients demand the product, and so on. These requirements need the predefined IP to be modified without restarting the system.

In this paper we propose a support for managing interactions between agents in terms of conversations. The term conversation is used to express every possible sequence and combination of messages that can be passed between two or more agents participating in a given agent system [18]. We provide a framework integrated into the Magentix Multiagent Platform (MAP) for defining conversations by means of IPs. Furthermore, these conversations can be dynamically modified in order to adapt the agent behaviour to new requirements.

The remainder of the paper is organized as follows. Section 3 introduces the Magentix MAP. Section 2 shows some previous related works. Section 4 presents the support implemented in Magentix for agent IPs. Section 5 presents an example that uses these dynamic IPs. Finally, section 6 presents some concluding remarks.

2. Related Work

IPs are broadly used for specifying patterns of communications between agents. Some MAPs provide support for executing IPs [2,3,4,5,1]. These implementations usually take the FIPA approach [15] in which an interaction is represented by a sequence of messages exchanged between an initiator and a participant agent role. Nevertheless these functionalities only consider predefined IPs and therefore specifications cannot be changed at execution time. Our approach is to allow dynamic changes in these IPs.

The goal of McGinnis and Walton[17,22] is to provide a support for defining IPs at run-time. Therefore agents are able to interact in systems where the IP may be unknown beforehand. They provide a language for defining IPs which will be created during the conversation by the participants. Although this proposal focuses on large and open MAS, it is not integrated in any MAP. This severely limits its practical application. Our solution provides a support for dynamic IPs at the MAP level. Therefore, we extend the Magentix MAP functionality to support the management of not only single messages but also entire conversations. This low level functionality does not restrict the applications to being built using specific models.

Artikis et al. [8,7] present a framework for specifying open systems from the perspective of organizations instead of individual agents. They represent open MAS as normative systems by specifying what is permitted, prohibited and obligatory. In this framework the specification of protocols is carried out in design time but can be modified in execution, due to the fact that rules which govern the protocol may change. However this view restricts the range of applications to normative systems. We propose a support at the MAP level that is independent of the MAS developed on top of it.
According to the previous works, our proposal tries to provide a MAP level support for managing complex interactions between agents that can change dynamically.

3. Magentix Multiagent Platform

Computing infrastructures supporting MAS focus on three levels [16]: (i) organization level: technologies and techniques related to agent societies; (ii) interaction level: technologies and techniques related to communications between agents; and (iii) agent level: technologies and techniques related to individual agents (such as reasoning and learning).

Magentix is aimed at supporting and enabling the development and execution of open MAS. It focuses on providing support at organization and interaction levels, which are key levels in open environments where heterogeneous agents interact and organize themselves into Virtual Organizations.

On the one hand, agent organizations have associated norms according to what a member is permitted to do, prohibited from doing and is obligatory according to her role and relationships with other members. On the other hand, interactions between agents are carried out by IP specifications which represent the communication rules.

3.1. Organization Level

Magentix implements a service-oriented approach for providing support to Virtual Organizations. Therefore, systems where participants are dynamically entering and leaving the system can be developed. The organization level is based on the THOMAS architecture [19]. The THOMAS main components are:

- Service Facilitator (SF) which allows the registration and search of services provided by external entities by following Service Oriented Architectures guidelines.
- Organization Management System (OMS) which is in charge of Virtual Organizations management, taking control of their underlying structure, the roles played by the agents and the norms that govern the system behaviour.

In this paper, we focus on detailing the interaction level support that Magentix provides. For a thorough explanation of THOMAS please see references [19].

3.2. Interaction Level

The support that Magentix provides for agent interactions is composed of two main parts: agent communication and, agent conversations. Magentix uses the Advanced Message Queue Protocol (AMQP) standard [21] as a foundation for agent communication. This standard supports asynchronous reliable message exchanges and facilitates the interoperability between heterogeneous entities. Magentix allows heterogeneous agents to interact with each other via messages following the FIPA-ACL [12] standard, that is exchanged using the AMQP standard. Magentix uses the Apache Qpid implementation of AMQP. This implementation provides QPid Client APIs for several programming languages. Therefore, although the current implementation of Magentix provides an API to develop agents written in Java, this could be extended to write agents in different lan-

[^3]: http://qpid.apache.org/
guages such as C++, Java, C# .NET, Ruby and Python. What is more, any proprietary im-
plementation that follows both AMQP and FIPA-ACL standards would be interoperable
with Magentix agents. Magentix also allows messages to be sent using the FIPA-HTTP
message transport protocol.

Interactions among agents in Magentix are oriented to conversations. A conversa-
tion between agents is represented by a sequence of messages following a specific IP.
Magentix facilitates the specification, automatic execution and management of every IP
which an agent is carrying out. IPs can be modified over time according the requirements
of the system. This support is further described in section 4.

4. Magentix Agent Conversations

As has been stated in [15], ongoing conversations between agents often fall into typical
patterns. In such cases, certain message sequences are expected and, at any point in the
conversation, other messages are expected to follow. These typical patterns of message
exchange are called IPs. In Magentix each conversation between agents represents a
sequence of messages which correspond to a specific IP.

According to FIPA specifications, an IP is generally represented by two agent roles
which represent different behaviours: initiator, which corresponds to the agent who ini-
tiates the IP, and participant, which corresponds to the agent (or agents) who participates
in the IP. Therefore, the sequence of messages regarding an IP are exchanged between
agents who play these roles.

As an illustration we are going to show a common real world situation: a cab driver
and a customer. When the customer enters a cab she does not have to worry about all
possible statements that might be made about addresses. Instead, the cab driver expects
to be told about an address and later, when the cab arrives at that address, the customer
must pay the driver. The expected sequence of messages in this scenario represents a
specific IP, that we may call Cab IP. The cab driver and the customer play different roles
in the conversation. In this example, the customer is the initiator and the cab driver is
the participant.

From the agent paradigm point of view, the cab driver could attend to more than a
single customer request simultaneously. Therefore, there would be several conversations
executing the Cab IP between the cab driver and several customers.

We provide an API to help programmers to develop IPs. This support allows pro-
grammers to create conversations following IPs and to change these protocols at execu-
tion time. In the following sections we explain the components of the API and how they
work.

4.1. Conversation Processors

As we said in a previous section, a conversation represents a sequence of messages be-
tween an initiator and one (or more) participant. Depending on the role that an agent is
playing in the conversation, specific actions are carried out in each step of the conversa-
tion. A Conversation Processor (CProcessor) is in charge of performing the actions and
managing the messages sent and received for each step of the conversation.

Depending on the role that an agent is playing, specific actions may carried out at
any given point of the conversation. Furthermore, specific actions are allowed to follow
other actions. In order to represent this information, a \textit{CProcessor} uses a direct graph composed of nodes and arcs. Nodes represent all the possible states of the role during the conversation and arcs represent a transition function between nodes, that is, what states can be reached from any state of the conversation.

In our example a node could be the moment while the \textit{cab driver} waits for the \textit{customer} to tell her an address. An arc could represent the transition from the state in which the \textit{cab driver} is driving and the state in which she waits to be paid, once she has arrived at the address.

We define several states which represent the different actions that can be performed during a conversation:

- \textit{Begin}: This state represents that the role starts the conversation.
- \textit{Final}: This state represents that the role ends the conversation.
- \textit{Action}: This state represents any action performed other than a speech act. For example it could be an action requested during the conversation.
- \textit{Send}: In this state the role sends a message.
- \textit{Wait}: In this state, the conversation halts until a message is assigned to the conversation. Then, according to the type of the message received, a specific subsequent \textit{Receive} state is executed. The type of the message is defined by its header.
- \textit{Receive}: This state must be preceded by a \textit{Wait} state. In this state the role receives a message. Each \textit{Receive} state manages messages with a specific set of headers.
- \textit{Initiate}: In this state the role starts a new subconversation, further explanation will be given in section 4.3.
- \textit{Participate}: This state is a special type of receive state where the role starts a new subconversation when receives the appropriate message, as well as the initialize state, it will be explained in section 4.3.

Therefore, IP can be defined in terms of these states and transitions. In the \textit{Cab IP}, the graph associated to the \textit{cab driver} agent role is shown in Figure 1 while the graph associated to the \textit{customer} agent role is shown in Figure 2.

![Graph for the cab driver agent role (participant) in the cab IP](image)

The graph associated to a \textit{CProcessor} can be dynamically modified at execution time. Modifications can indistinctly refer to states or arcs. Therefore, a \textit{CProcessor} can adapt itself to any kind of change while the IP is being executed. For example, from now on, the fare will be calculated before travelling and the \textit{customer} has to pay in advance. This change in the protocol can be done dynamically. Specifically, the \textit{send} state asking for the money, will be moved right after receiving the request message with
the destination address. The new graph directing the conversation protocol is shown in Figure 3. It is worth noting that a change in the IP of one of the conversation roles may involve a change in the IP of the other role. In our example the protocol associated with the customer should be modified according to the modifications on the protocol associated to the cab driver; for the sake of brevity we do not include the figure showing these changes.

4.2. Conversation Factories

Conversation factories (CFactories) are in charge of starting conversations. A CFactory has a kind of CProcessors associated which can manage a specific IP. When a new conversation must be created the specific CFactory starts a CProcessor which will be responsible for executing this conversation.

There are two types of CFactory; when to use each one depends on the role that the agent plays in the conversation. If the agent plays the initiator role in the conversation, she has to use an initiator CFactory. On the other hand, if the role of the agent is that of participant, she has to use a participant CFactory. An initiator CFactory starts a new conversation without requiring any external event. A participant CFactory starts a new conversation when an appropriate message arrives for the agent. In order to decide if a message requires a CProcessor to start, participant CFactories use a message filter. This message filter specifies values of the message header fields. When an agent receives a message its header is checked by the participant CFactories in order to find which one is responsible for starting the CProcessor.

Each conversation has a specific conversation_id which is assigned by the CProcessor of the initiator role of the conversation. This conversation_id is used to associate each message with its corresponding conversation. If the message has the same conversation_id of an already running conversation of the agent, the message is delivered to the corresponding CProcessor without being checked by the CFactory.

In our example, the cab driver could be seen as an agent with one participant CFactory. The filter of the CFactory would only accept messages with the performative request. The customer could be seen as an agent with one initiator CFactory with a CProcessor, which in its first state would send a message with the performative header set to request, and the content specifying the destination address.

4.3. Nested Conversations

Conversations may need to start other conversations called sub-conversations. Therefore, the IP corresponding to the main conversations may be composed of sub-protocols. Our API offers two different types of sub-protocol: synchronous and asynchronous. In a synchronous sub-protocol, the parent conversation waits until the sub-conversation ends in order to resume. In an asynchronous sub-protocol, the parent conversation does not wait for the sub-protocol to finish.
There are two special conversation states meant to start sub-conversations: *initiate* and *participate*. On the one hand, *initiate* starts sub-conversations as initiator when the CProcessor reaches that state. On the other hand, *participate* starts sub-conversations after receiving an appropriate message. This is the same mechanism used by a participant factory to start new conversations.

When a synchronous sub-conversation is initiated, it is assigned the same *conversation_id* of the conversation that started it. If the sub-conversation is asynchronous, it is assigned a new *conversation_id*.

Figure 4 shows the cab example modelled using sub-protocols. The state "send: request for the fare money" is an initialize state that starts a synchronous sub-protocol. This sub-protocol is the FIPA Request IP[14], as shown in Figure 5. In this sub-protocol, the cab driver is the initiator (he asks for the money) and the customer is the participant (he receives the message requesting the money). It is worth noting that the customer agent may also be modified. In this way, a participate state may be needed in order to respond to the FIPA Request sub-protocol initiated by the driver.

### 4.4. Magentix Agent View

Once all the components of the API have been explained, it is possible to show a global view of a generic Magentix Agent with several *CFactories* and holding several conversations simultaneously. Figure 6 shows a Magentix Agent with two participant *CFactories*. *Participant Factory 1* starts a new conversation when a message with a new *conversation_id* and *inform* performative is received. *Participant Factory 2* starts a new conversation when a message with a new *conversation_id* and *request* performative is received. In the figure the agent is holding two conversations created by *Participant Factory 2*. Any further message related with any of these two conversations will be directly assigned to the corresponding conversation. The agent has also two initiator *CFactories*. The *initia-
tor factory 1 has created the conversation with the conversation_id 1. This conversation is held simultaneously with the other two conversations.

5. API Example

In this section a source code example is shown. This example is based on the cab example and explains how the cab driver agent could be programmed using the Magentix API. The IP is the one shown in figures 1 and 2.

When programming a Magentix agent, the user must implement two methods: Initialize and Finalize. In the Initialize method the user normally defines the IP that the agent will use during its existence by means of defining CFactories and CProcessors. The Finalize method is executed when the agent finishes its activity on the platform.

In order to program the cab driver we have to add the CFactory that creates the CProcessor that attend the requests from the customers.

```java
protected void Initialize(CProcessor myProcessor, ACLMessage welcomeMessage) {
    ACLMessage template;
    template = new ACLMessage(ACLMessage.REQUEST);
    CFactory talk = new CFactory("TALK", template, 1, this);
    ......
    qje3
}
```

In line 4, we create a CFactory called talk, specifying its name, the message template that it will use as a filter (in this example we only want messages with Request performative to be managed by this CFactory), how many conversations of this IP this agent can attend at the same time (in this example only one) and the agent who will own the CFactory (in this case the CabDriver agent).

Once the CFactory is created we have to create the template of the CProcessors that this CFactory will produce. Firstly, we have to specify the graph and each one of its states. Basically, for each state we specify a method that will be executed when the IP reaches that state. An exception to this is the wait state, which just waits for a message to arrive to the CProcessor and has no method to execute. Each method returns the name of the next state that the CProcessor has to execute. This is done as shown in the following code:
protected void Initialize(CProcessor myProcessor, ACLMessage welcomeMessage) {
    ......
    //BEGIN state
    BeginState BEGIN = (BeginState) talk.cProcessorTemplate().getState("BEGIN");
    class BEGIN_Method implements BeginStateMethod {
        public String run(CProcessor myProcessor, ACLMessage msg) {
            return "WAIT";
        }
    }
    BEGIN.setMethod(new BEGIN_Method());
    //WAIT state
    talk.cProcessorTemplate().registerState(new WaitState("WAIT", -1));
    talk.cProcessorTemplate().addTransition("BEGIN", "WAIT");
    //RECEIVE state
    ReceiveState RECEIVE = new ReceiveState("RECEIVE");
    class RECEIVE_Method implements ReceiveStateMethod {
        public String run(CProcessor myProcessor, ACLMessage messageReceived) {
            CabDriver myAgent = (CabDriver) myProcessor.getMyAgent();
            myAgent.destination = messageReceived.getContent();
            return "ACTION";
        }
    }
    RECEIVE.setAcceptFilter(new ACLMessage(ACLMessage.REQUEST));
    RECEIVE.setMethod(new RECEIVE_Method());
    talk.cProcessorTemplate().registerState(RECEIVE);
    talk.cProcessorTemplate().addTransition("WAIT", "RECEIVE");
    //ACTION state
    ActionState ACTION = new ActionState("ACTION");
    class ACTION_Method implements ActionStateMethod {
        public String run(CProcessor myProcessor) {
            CabDriver myAgent = (CabDriver) myProcessor.getMyAgent();
            myProcessor.myAgent.driveToDestination(myAgent.destination);
            return "SEND";
        }
    }
    talk.cProcessorTemplate().registerState(ACTION);
    talk.cProcessorTemplate().addTransition("RECEIVE", "ACTION");
    //SEND state
    SendState SEND = new SendState("SEND");
    class SEND_Method implements SendStateMethod {
        public String run(CProcessor myProcessor, ACLMessage messageToSend) {
            CabDriver myAgent = (CabDriver) myProcessor.getMyAgent();
            messageToSend.setPerformative(ACLMessage.REQUEST);
            messageToSend.setContent(myAgent.calculateFare());
            messageToSend.setReceiver("customer");
            return "FINAL";
        }
    }
    talk.cProcessorTemplate().registerState(SEND);
    talk.cProcessorTemplate().addTransition("ACTION", "SEND");
    // FINAL state
    FinalState FINAL = new FinalState("FINAL");
    class FINAL_Method implements FinalStateMethod {
        public void run(CProcessor myProcessor, ACLMessage messageToSend) {
            messageToSend.setContentType("DONE");
        }
    }
    FINAL.setMethod(new FINAL_Method());
    talk.cProcessorTemplate().registerState(FINAL);
    talk.cProcessorTemplate().addTransition("SEND", "FINAL");
    ......}

6. Conclusions
This paper presents a support for managing interactions between agents in terms of conversations. This support is provided by the Magentix MAP and it is based on the spec-
ification and execution of dynamic IPs on top of an interoperable agent communication mechanism.

IPs are specified as graphs and new conversations that follow these IPs are created by CProcessors and CFactories. This is a first attempt towards supporting dynamic agent interactions which are needed in open MAS that are highly changeable environments.

7. Acknowledgments

This work has been partially supported by CONSOLIDER-INGENIO 2010 under grant CSD2007-00022, and TIN 2008-04446 and PROMETEO/2008/051 projects. Jose M. Such is enjoying a grant from Conselleria d’Empresa, Universitat i Ciencia de la Generalitat Valenciana (BFPI06/096). Juan M. Alberola is enjoying a grant from Ministerio de Ciencia e Innovación de España (AP2007-00289).

References