An Agent Deployment Model Based on Components*

Fabio Melo
fabio@inf.puc-rio.br

Ricardo Choren
choren@inf.puc-rio.br

Renato Cerqueira
rcerq@inf.puc-rio.br

Carlos J. P. de Lucena
lucena@inf.puc-rio.br

Marcelo Blois
Faculdade de Informática, PUCRS, Rio Grande do Sul, Brasil
blois@inf.pucrs.br

PUC-RioInf.MCC 37/03 October, 2003

Abstract: In the past few years, the Multi-agent systems (MAS) area has presented an accelerated growth. New techniques and tools are constantly being proposed and several methodologies have been published to support the development of MAS. Most of these methodologies concentrate on the system analysis phase, giving almost no support for MAS implementation. Since agents can be seen as sort of specialized distributed components, in this paper we propose an agent deployment model based on the CORBA Component Model. We also describe a case study to show the potential of an agent-based application development using this model.

Keywords: multi-agent systems, components, implementation model, CORBA

Resumo: Ultimamente, a área de Sistemas Multi-agentes (SMA) está em crescimento. Novas técnicas e ferramentas estão constantemente sendo propostas e diversas metodologias têm sido publicadas a fim de oferecer suporte ao desenvolvimento de SMA. A maioria destas metodologias se concentra na fase de análise, deixando a fase de implementação em um segundo plano. Uma vez que agentes podem ser vistos como uma espécie de componente distribuído especializado, este trabalho propõe um modelo de implementação de SMA baseado no CORBA Component Model. Também é descrito um estudo de caso para mostrar o potencial do desenvolvimento de uma aplicação baseada em agentes usando este modelo.

Palavras-chave: sistemas multi-agentes, componentes, modelo de implementação, CORBA

* Sponsored partially by the PRONEX Project under grant 664213/1997-9 and by the ESSMA Project under grant 552068/2002-0.

1. Introduction

The term large-scale application applies to a class of applications that perform important business functions, such as automating key business processes. Usually, a large-scale application is distributed and is composed of several parts implemented with different types of technologies.

Multi-agent systems (MAS) development has promoted new ways of building large-scale applications that foster features such as autonomy, modularity, openness and distribution [12]. Software agents offer great promise as an abstraction that can be used by developers as a way to understand, model and develop software that operates in adaptive, flexible, large-scale environments [30].

Agent-oriented software design and programming [22] decomposes large distributed systems into autonomous agents, each driven by its goals. This decomposition is done with specification languages such as AUML [20], MESSAGE [5], AORUML [28], ANote [6] and MAS-ML [23] and it helps developers create a set of agents that collaborate among themselves, using structured high-level messages to achieve the system's goals.

After modeling a multi-agent system, developers face the challenges of implementing it. Despite the evolution of new platforms such as [4] [25] [27], currently there is no widely accepted agent-oriented language or architecture to support agent-based software development. Thus developers need to look for other ways to transform an agent system specification into a concrete robust implementation.

Software components show much promise at satisfying the demands of inherent modularity on which agents need to be added and recombined to form new applications. Moreover, component middleware, such as the CORBA Component Model (CCM), J2EE and the emerging web services middleware (.NET and ONE) have shown great potential for the development of improved component-based solutions.

Agent-oriented software development can extend the conventional component development, offering more flexibility, less design and implementation time and less coupling between agents [13]. In fact, an agent can be seen as a complex component, composed of several simpler ones.

In this paper, we address some of the above issues and our work in progress on creating an implementation model to build scalable, large-scale, distributed agent systems using a component model. The rest of the paper is structured as follows. We begin with a review of components, middleware and how they impact on agent-based development. Section 3 describes the CCM in more detail. Section 4 presents a brief description of the proposed Agent Deployment Model. Section 5 presents a sample case study. Section 6 discusses some related work. We conclude in Section 7.

2. Components, Middleware and Agents

A software component is defined as a self-contained, customizable and large-grain building block for (possibly distributed) application systems [26]. Some of the common characteristics of components are their ability to communicate by sending and receiving messages, the possibility of having multiple components of the same type in a system, and the description of their state consisting of attributes and other aggregated components [29].

Components encapsulate specific services or sets of services to provide reusable building blocks that can be composed to develop large-scale applications more rapidly and robustly than those built entirely from scratch. Component-based software engineering produces a set of reusable components that can be combined to obtain a high-level of modularity and reuse while developing applications [14].

Middleware is reusable software that resides between the applications and the underlying operating systems, network protocol stacks and hardware [21]. It works as a bridge between application programs and the basic lower-level hardware and software infrastructure that deals with how the parts of these applications are connected and interoperate.

From the development point of view, a middleware decouples application-specific functionality from the inherent complexities of the infrastructure. Thus it enables developers to concentrate on programming application-specific functionality instead of being concerned about low-level infrastructure problems. Several technologies, such as CORBA, .NET and J2EE, emerged to decrease the complexities associated with developing distributed large-scale software. Their successes have been a key factor to the spread of the middleware paradigm.

Components and middleware can be used to develop agent-oriented software. Some advantages [13] include:

– The compliance of components to message-oriented rather than method-oriented communication.

– The creation of more flexible, adaptable, robust and self-managing components yields the combination of agents and services.

– Dynamic registration and discovery of components by name and features, and the loose coupling lead to a greater degree of independent development. For instance, agents can be loaded and activated while the system is running.

– The exploitation of technologies such as patterns, micro-frameworks and generators to create multi-agent protocols and behaviors.

Components, as agents, are service oriented and should be choreographed (coordinated), leading to the creation of loosed-coupled organizations. The use of components can exploit established technologies such as UML [3], design patterns [11], frameworks [7] and aspect-oriented development [15] for multi-agent system design. Also, agent component systems can be built on some emerging agent infrastructures such as ZEUS [4], JADE [27] or on current standard infrastructures as J2EE [24], .NET [16] and CORBA [19].

3. The CORBA Component Model

CORBA Component Model (CCM) [17] is a server-side component model for building and deploying CORBA applications. It uses accepted design patterns and facilitates their usage, enabling a large amount of code to be generated and allows system services to be implemented by the container provider rather than the application developer.

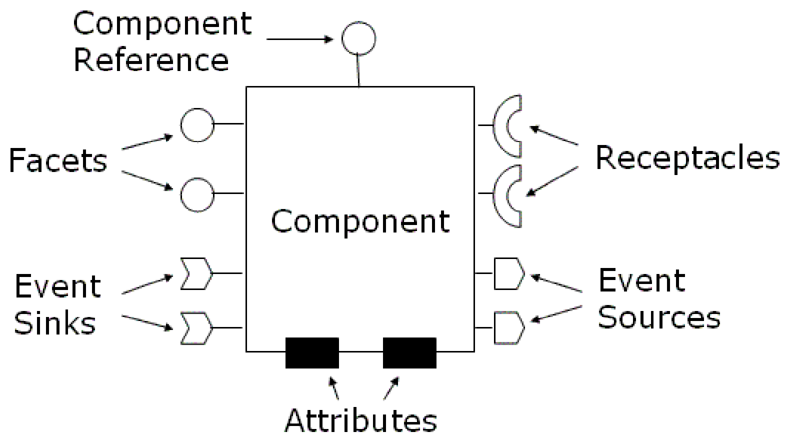

Figure 1. CORBA component

The CCM specification introduces the concept of component and the definition of a comprehensive set of interfaces and techniques for specifying implementation, packaging and deployment of components [1]. A component is a basic meta-type in CORBA. The component meta-type is an extension and specialization of the object meta-type [18]. As seen in Figure 1, components support four kinds of ports through which clients and other elements of an application may interact. They also support attributes, which represent their properties.

3.1. Facets

The component facets are the interfaces that the component exposes. Facets are object references for the many different interfaces a component may support. The component has a main reference and may have many facets, supporting distinct interfaces and representing new references. It is possible to navigate from the component's main reference to a facet's reference and vice-versa.

The implementation of the facet interfaces are encapsulated by the component and considered to be parts of the component. The internal structure of a component is opaque to clients. The life cycle of a facet is bounded by the life cycle of its owning component.

3.2. Receptacles

Component systems contain many components that work together to provide the client functionality. The relationship between components is called a connection and the conceptual point of connection is called a receptacle. The receptacle allows a component to declare its dependency to an object reference that it must use. Receptacles provide the mechanics to specify interfaces required for a component to function correctly.

A component may exhibit zero or more receptacles. A receptacle may be able to accept only one object reference (simple receptacles) or many references (multiple receptacles). A multiple receptacle may choose to limit the number of simultaneous connections allowed. Operations supported by the receptacles are: connect and disconnect operations, and return connections.

3.3. Event Sources and Event Sinks

Allow components to work with each other without being tightly linked, supporting the publish–subscribe event model. This is loose coupling as provided by the Observer design pattern. A component is an event source when it declares its interest in publishing or emitting an event. The difference between publishing and emitting is that a publisher is an exclusive provider of an event while an emitter shares an event channel with other event sources.

Other components become subscribers or consumers of those events by declaring an event sink. Components that want to be notified when a certain event occurs must connect its event sink to the event source of the component where the event might occur. When the event is generated, the component notifies all components that connected to its event source.

3.4. Attributes

An attribute is an extension of the traditional notion of CORBA interface attributes that allow component values to be configured. The CCM version of attribute allows operations that access and modifies values to raise exceptions. It means that an attribute in a component only supports the read/write operations. This is a useful feature for raising a configuration exception after the configuration has completed and an attribute has been accessed or changed.

4. The Agent Deployment Model

A MAS is a system composed of agents. Primarily an agent is as an element that can perceive limited aspects of its surrounding application and that can act in this application, either directly or through the interoperation with other agents. Thus, an agent can be perceived as a subsystem that has a reference (identity), an interface for sending and receiving messages (interaction), a set of plans (action) and a core (reasoning and state management).

The agent deployment model (ADM) we propose combines component and middleware technologies for the development of agent-based systems. The model (figure 2) shows a generic agent composed of two basic components: Agent and Plan components. These components should be seen as the agent's implementation skeleton. Each agent type that will be effectively implemented shall extend Agent and Plan generic components to implement its specific features.

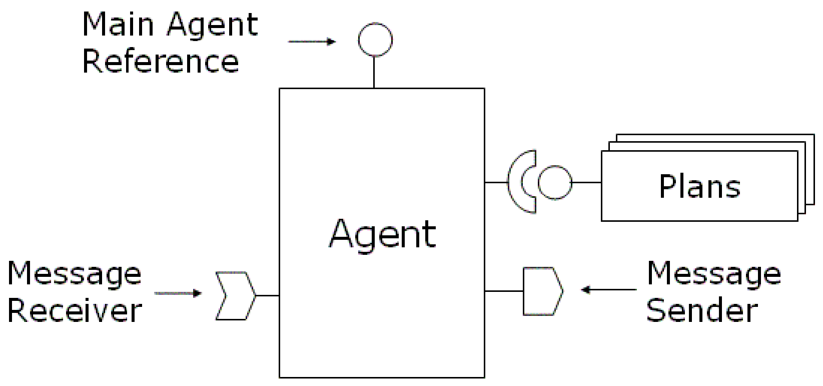

Figure 2. The Agent Deployment Model

The Agent component is the body of the agent. It has an interface that holds the main reference that identifies the agent as a whole. This component also stores the agent's data as attributes. It is responsible for the entire agent reasoning, such as properly dealing with message reception and sending and checking its current state in order to manage action plans (starting, stopping, resuming, canceling, etc).

The agent's action plans are described as sub-components accessed through receptacles. This allows for agent evolution and adaptation. If an agent evolves or adapts, it can dynamically add, change or remove action plans from its plan component library.

Developing action plans as components allows for the reuse of agent functionality. An agent offers a service through a set of action plans. If a multi-agent system requires two or more agents to provide a same service, the developer does not need to write code again, nor use copy and paste. The developer deploys an action plan component and configures the receptacles of the agents who will deliver the service to that component.

Conversations between agents are asynchronous. Agents send and receive messages and they decide autonomously whether or when a message should be processed. An agent may receive a message and decide to answer it later or even ignore it. Thus it is not recommended to map the agent's communication channels to facets and receptacles since these ports are not indicated for an asynchronous communication.

Event senders and receivers implement the interaction features of an agent. There must be an event sender for any message an agent has to send (from the system specification). Likewise, there must be an event receiver for every message an agent has to receive.

The publish-subscribe model of communication allows for the use of any agent communication language. Developing the agent as a set of components also eases the question of learning other communication languages. For instance, if an agent currently understands a message in a format A and it needs to understand a new format B, it will be necessary to add a new component for format B to the agent and some little reengineering effort on the main agent component.

The ADM offers a model for a single agent. However, multi-agent systems gather agents in organizations. An organization defines a boundary around a set of agents, delimiting the set of services it provides. Since an organization also provides services, it can be seen as an agent, at a higher level. So in ADM, an organization is developed as an agent (with an agent component) that has receptacles for the set of agents it encompasses. The agent component identifies the organization and should be responsible to be the communication bridge between the agents inside the organization and the other agents of the multi-agent system.

It is important to mention that the ADM does not classify agents functionally as, for instance, administrative agents, matchmakers or yellow pages. Administrative functions such as agent registration and discovery are directly dealt with by the middleware infrastructure since components already need to register and find themselves. Every agent in the ADM is viewed as a task agent that performs services and that produces and consumes messages as required. Thus, the ADM does not deliver any kind of agent beforehand.

A multi-agent system is not only composed of agents. It has other non-agent components such as knowledge resources, databases, and other objects that describe the system's ontology (agent knowledge). The ADM does not make any special assumptions about the non-agent components of a multi-agent system. These should be developed as components that will be used by agent components.

As shown in figure 2, the ADM describes an agent similar to a CORBA component defined in the CCM. The use of CCM as a base for the ADM is not coincidental. The model itself encompasses the basic agent characteristics: identity, interaction, action planning (pro-activity and autonomy) and reasoning. Moreover, the advantages (code generation and less application developer implementation) can be used to speed up agent-based systems development.

The CCM defines features and services in a standard environment, enabling the implementation, management, configuration and deployment of components that integrate with commonly used CORBA services. These server-side services add other facilities to agent-based applications, including transactions, security, persistence and events.

5. Mapping and Implementation Case

In order to validate our approach, we have experimented with some agent-based specifications that are available in the literature. These include a coffee pot system [31], a simple negotiation system [2], and a PC manufacture supply chain [4].

In addition, we have conducted one medium-sized case study. The case study involved specifying an agent marketplace. We first constructed a system specification using ANote in order to specify the agents' goals, plans and interactions. Then we were able to validate the ANote agent models obtained from our specifications against the ADM. From the case study, we have been able to increase our confidence in the ADM as a means for constructing agent-based systems.

5.1. An Agent Marketplace

In this section we present an agent-based marketplace where software agents interact to buy and sell products, according to the proposals informed by its users. Each agent will be assigned a proposal, whether it is a purchase or a sale proposal. In this marketplace, it is possible to create groups to cooperatively buy products.

The first step of modeling a multi-agent system with ANote is to define the system's goals and the agents that will carry out these goals. The system's goals build a hierarchy in order to define the functional goals of agents. These functional goals will be the basis for the development of action plans.

In this example, three types of agents were defined: buyers, sellers and managers. The buyer main goal is to buy a product; the seller main goal is to sell a product; and the manager main goal is to manage buyer groups. Some other elicited functional goals include: dealing with other agents, advertising and searching for proposals, creating purchase groups, joining an existing group and inviting buyers to join a purchase group. Hereafter, we will be showing the use of the ADM for the development of a manager agent.

The manager agent should be initially described as a specialization of the ADM agent component, called Manager, which will be the base for all the manager agents in the system. Continuing with the system modeling with ANote, it is now necessary to specify the contexts in which the agent will achieve its goals. This specification derives from the agents' action plans and interactions.

For each action plan of the manager agent, there should be specialized the ADM action plan component. We will call a manager action plan a Manager_Plan. Every Manager_Plan must have a receptacle to store the reference of the Manager it belongs. Figure 3 shows the preliminary mapping of the manager agent to the ADM.

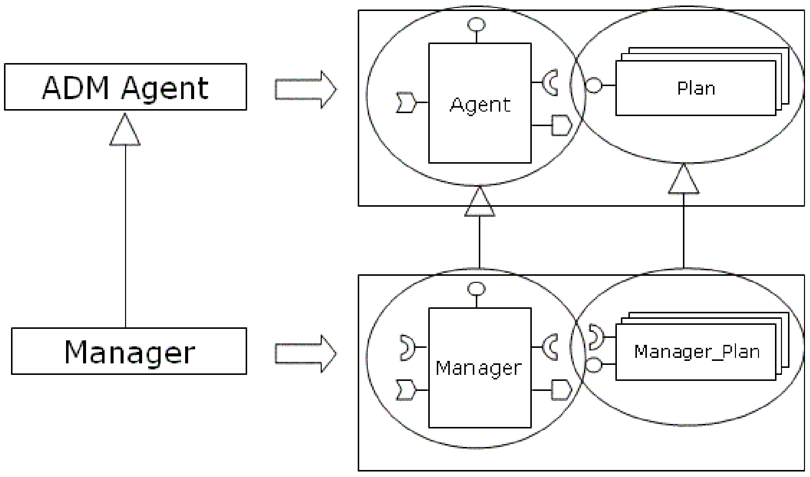

Figure 3. The manager agent in ADM

Furthermore, since the manager agent manages groups of buyers, the Manager component should have an extra receptacle to hold references from the group's buyers.

The plans identified for each agent are mapped directly to components with the plan type of the agent. Figure 4 shows a Manager's plan being mapped to a component of the Manager_Plan. Once the component is created, its reference is connected to the plans receptacle of the Manager component.

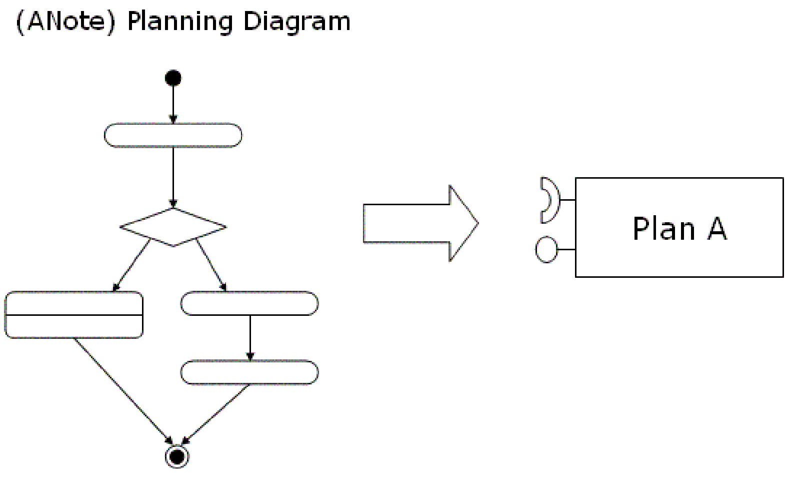

Figure 4. A plan in ADM

The interactions between agents are carried out by messages. Messages are mapped to a new CCM event called Message. In this example, this event was structured according to the definition of agent messages proposed by FIPA-ACL [9] [10]. A Message event contains all the message data. A message is transmitted when the sender agent notifies the receiver agent that a Message event has occurred. These notifications are made through the event source and sink ports (figure 5). The Message event is passed as a parameter of this notification.

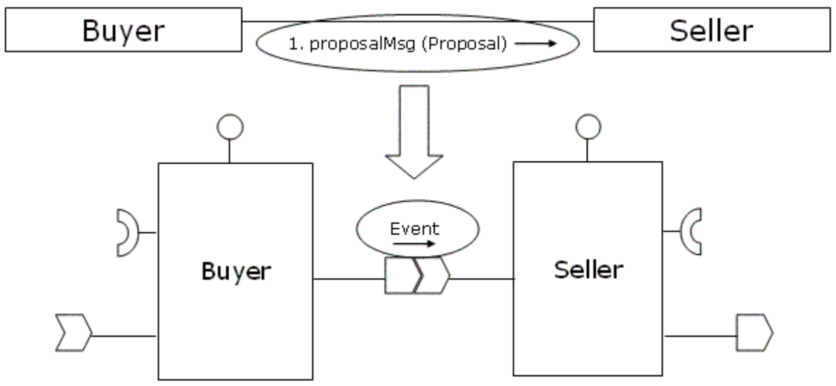

Figure 5. A message in ADM

This concludes the development of the manager agent into modular components. The other agents (buyer and seller) are developed the same way. It is also necessary to develop the non-agent elements that compose the multi-agent application. These elements are mapped to CORBA interfaces or structs (figure 6). The mapping is based on one simple rule. If the element presents data and behavior, it is mapped to a CORBA interface. Behaviors are mapped to operations and the data are mapped to interface attributes. If the element works only as a data repository and it does not have specific behavior, it is mapped to a struct, where its attributes represent the element's data.

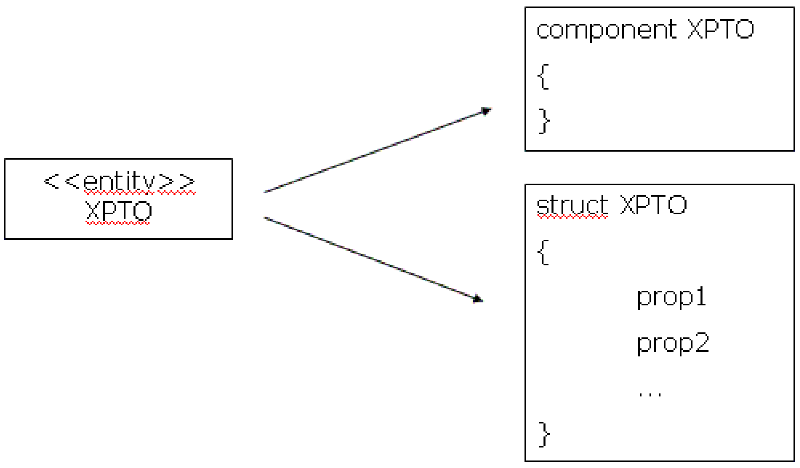

Figure 6. Non-agent Elements in ADM

In the marketplace example, non-agent elements include proposals, advertisements and products. The proposal element was mapped to a struct with the same name. The struct should hold all the element’s attributes such as proposal_type, product, quantity and price. The proposal element was mapped to a struct because it did not need operations; it only needed to store data.

To develop the system organizations, it is necessary to create other agent components to hold the references of the organization members. In the marketplace example, a MarketPlace agent component was created to define an organization of a Manager and the Buyers that compose a group.

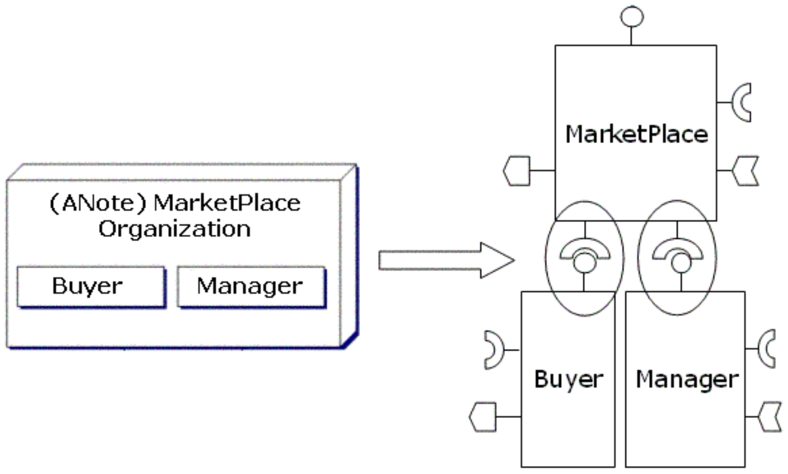

Figure 7. An organization in ADM

5.2. Implementation

This section discusses some implementation of our case study. The description is intended to help the reader understand the work involved in implementing an agent from the ADM component model.

Every agent in the system solution must have an agent component and a set of action plan components. A system solution agent component extends a basic Agent component and implements a basic AgentInt interface (figure 8). The AgentInt interface creates the features responsible for the agent lifecycle such as starting the agent, message dispatching and plan initiation. The Agent component is a template that defines that an agent uses multiple action plans and it is able to publish and consume messages.

```
interface AgentInt {
  agent_name get_name ();
  void send_msg (in Message msg);
  PlanInt get_plan (in string plan_name)
          raises (PlanNotFound);
  void start_agent ();
};

component Agent {
  uses multiple PlanInt plans;
  publishes Message emitter;
  consumes Message receptor;
};
```

Figure 8. Interface and component for agents

The Manager agent, from the example, looks like the following (figure 9).

```
interface ManagerAgentInt : AgentInt {
  ...
};

component ManagerAgent : Agent
          supports ManagerAgentInt {
  ...
};
```

Figure 9. The Manager agent definition

A plan component extends a basic Plan component and implements a basic PlanInt interface (figure 10). The PlanInt interface has only a definition to actually execute the plan method. The plan component must also have a reference for the agent component that uses it.

```
interface PlanoInt {
  AnySeq execute_plan (in AnySeq param);
};

component Plano supports PlanoInt {
};

component Manager_Plan : Plan {
  uses ManagerAgentInt agent;
  ...
};
```

Figure 10. Interface and component for plans

A message is an event type that, in the case study, implements the FIPA ACL message. Thus, it should define all the parts of an ACL message. The non-agent elements should also be defined. From the example, the proposal element was described as a struct (figure 11).

```
eventtype Message {
  public string performative;
  public string sender;
  public string receiver;
  public string reply_to;
  public string content;
  ...
};

struct Proposal {
  string proposal_type;
  string product;
  float price;
};
```

Figure 11. Message and non-agent definitions

6. Related Work

Several agent-based infrastructures have been proposed and, also, a number of techniques for building agents based on these infrastructures have been developed. Sycara and her group have developed RETSINA [25], a MAS infrastructure composed of a set of services and conventions that allow agents to interoperate. It provides a layered model in which its various components, such as a Communication Infrastructure, are organized. So, in RETSINA, the components are part of the platform, which is different from the ADM - the components build the agents, and the platform is the component middleware. This does not allow the easy reuse of components of agents such as action plans.

The RETSINA infrastructure supplies a MAS Management Service to provide agent registration and discovery. In ADM, this is done directly by the component middleware, requiring no extra programming effort. RETSINA also deals with agent security, a major issue in mobile agents. In ADM this is also left to the component middleware; however, since the used component middleware (CORBA) is not meant for agents, some security issues such as authenticity are not present in ADM.

ZEUS [4] is a toolkit for constructing collaborative multi-agent applications. The ZEUS toolkit consists of a set of components that can be categorized into three functional groups (or libraries): an agent component library, an agent building tool and a suite of utility agents comprising nameserver, facilitator and visualize agents.

The component library is composed of planning, communication, social interaction, data structures and user interface components. Planning, communication and social interaction components are also supported by ADM. However, the ZEUS toolkit is not open, not offering support for the maintenance, change and addition of these components. For instance, ZEUS offers scheduling and coordination engine components, but a developer can only use those provided by the toolkit.

JADE (Java Agent Development Framework) [27] is a software development framework aimed at developing multi-agent systems and applications conforming to FIPA standards for intelligent agents. It includes two main products: a FIPA-compliant agent platform and a package to develop Java agents. JADE has two main components: Agent and Behavior. The Agent component is the class that must be extended by application programmers to create agents, a mechanism similar to ADM. The Behavior components implement the tasks, or intentions, of an agent. They are logical activity units, similar to the ADM plan components, that can be composed in various ways to achieve complex execution patterns and that can be concurrently executed. However, JADE offers a limited set of Behavior components, reducing the reuse potential, such as ZEUS.

Unlike these works, ADM does not make any platform classification of components (e.g. communication, interaction). It only classifies components into agent capabilities so that these components can be more easily reused in different agent-based systems. Also, ADM does not classify agents (e.g. nameserver, facilitator). The idea is that specific agents developed using the component model build an agent library in order to create an agent reuse repository.

Another issue is the communication language. All the above works use only languages based on FIPA ACL, such as KQML [8], for agent communication. This feature is bound to the platform. In ADM, agent communication is done through a publish-subscribe mechanism. It does not predefine any agent communication language. This allows for extensibility, since other languages can be used, and reuse, since a component that deals with a language can be reused in other MAS that utilize this language.

7. Conclusions

We have presented a deployment model for multi-agent systems in terms of components. We have defined and implemented a set of components and illustrated, with an example, how they can be used to support the implementation of MAS. Using the model, we help to manage the complexity of MAS specifications, promoting agent reuse, and providing a simple, operational way of showing how the component technology relates to agent-based development. Using components to develop agents we help developers make an explicit correlation between domain-specific or general assumptions in agent specifications to an implementation model.

The model allows for the development of single agents and a way for them to work together. It provides a way to implement agent capabilities – core (to do reasoning and state management), action plans and interaction. Moreover, since it is based on a component middleware, it provides location services, communication platform and knowledge (ontology) development.

One direction for future work is to study how MAS specification languages and methods can be integrated into our approach. Use of a variety of languages for describing agent capabilities and system specifications should show us how to improve the current model. Another direction is that of experimenting with component platforms other than CORBA.

Finally, to help make multi-agent technologies adoption happen, there is a need to build agent platforms, models and implementation toolkits to ease the generation of individual agents and multi-agent systems. Components have proven to be a good technique for bridging the gap between agent specifications and agent implementations. We believe an important direction of future work should focus on building mechanisms to provide feedback to system designers about the implementation's world. It would be very desirable to automate the construction of some components of the agent system from synthesized agent models. This can allow designers to gain more insight into the specification and to show its impact in the actual system construction.

References

[1] D. Bartlett, "CORBA Component Model (CCM): Introducing next-generation CORBA", online (http://www-106.ibm.com/developerworks/webservices/ library/co-cjct6/), Apr. 2001.

[2] Bigus, J.P. and J. Bigus, Constructing Intelligent Agents Using Java: Professional Developer's Guide, 2nd edition, John Wiley & Sons, 2001.

[3] Booch, G., J. Rumbaugh, and I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1998.

[4] British Telecommunications plc, "BT Intelligent Agent Research", online (http://more.btexact.com/projects/agents/zeus/index.htm), Oct. 2002.

[5] G. Caire, "MESSAGE: Methodology for Engineering Systems of Software Agents Initial Methodology", Technical Report, EDIN 0224-0907, Project P907, EURESCOM, 2001.

[6] R. Choren, and C.J.P. Lucena, "An Approach to Multi-Agent Systems Design Using Views", Technical Report, Computer Science Department, PUC-Rio, 2003.

[7] Fayad, M.E., D.C. Schmidt, and R.E. Johnson (Eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, John Wiley & Sons, 1999.

[8] T. Finin, R. Fritzson, D. McKay, and R. McEntire, "KQML as an Agent Communication Language", Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, ACM Press, 1994, pp. 456-463.

[9] Foundation for Intelligent Physical Agents, FIPA ACL Message Structure Specification, online (http://www.fipa. org/specs/fipa00061), Dec. 2002.

[10] Foundation for Intelligent Physical Agents, FIPA Communicative Act Library Specification, online (http://www.fipa.org/specs/fipa00037), Dec. 2002.

[11] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, Design Patterns, Addison-Wesley, 1995.

[12] Garcia, A.F., C.J.P. Lucena, F. Zambonelli, A. Omicini, and J. Castro (Eds.), Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications, Lecture Notes in Computer Science 2603, ISBN 3-540-08772-9, Springer, 2003.

[13] M.L. Griss and R.R. Kessler, "Achieving the Promise of Reuse with Agent Components". In Garcia, A. et al (eds.), Software Engineering for Large-Scale Multi-Agent Systems, Lecture Notes on Computer Science, Springer-Verlag, 2002, p. 139-147.

[14] M. Griss and G. Pour, "Accelerating Development with Agent Components", IEEE Computer, 34(5), 2001, pp. 37-43.

[15] Kiselev, I., Aspect-Oriented Programming with AspectJ, SAMS, 2002.

[16] Microsoft Corporation, "Microsoft .NET", online (http://www.microsoft.com/net/), 2003.

[17] Object Management Group. CORBA Component Model, v 3.0, June 2002.

[18] Object Management Group. CORBA Components, v 3.0, June 2002.

[19] Object Management Group. Common Object Request Broker Architecture, v 3.0.2, Dec. 2002.

[20] J. Odell, H. Parunak, and B. Bauer, "Extending UML for Agents", Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, 2000, pp. 3-17.

[21] Schantz, R.E. and D.C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications". In Marciniak, J., and G. Telecki (eds.) Encyclopedia of Software Engineering, Wiley & Sons, 2002.

[22] Y. Shoham, "Agent-Oriented Programming", Artificial Intelligence, 60(1), 1993, pp. 139-159.

[23] V.T. Silva, and R. Choren, "Using the MAS-ML to model a Multi-Agent System", Technical Report 24/03, Computer Science Department, PUC-Rio, 2003.

[24] Sun Microsystems, Inc. Java 2 Platform Enterprise Edition Specification, v 1.3, Jul. 2001.

[25] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa, "The RETSINA MAS Infrastructure", Technical Report CMU-RI-TR-01-05, Robotics Institute Technical Report, Carnegie Mellon, 2001.

[26] Szyperski, C. Component Software: Beyond Object-Oriented Programming, Addison-Wesley, 1998.

[27] Telecom Italia Lab, "JADE Programmer's Guide", online (http://sharon.cselt.it/projects/jade/doc/programmersguide.pdf), Feb. 2003.

[28] G. Wagner, "Agent-Object-Relationship Modeling", Proceedings of the 2nd International Symposium: From Agent Theory to Agent Implementation, 2000.

[29] A. Wienberg, F. Matthes, and M. Boger, "Modeling Dynamic Software Components in UML", Proceedings of the Second International Conference on UML (UML'99), Fort Collins, Colorado, Springer-Verlag, 1999, pp. 204-219.

[30] Wooldridge, M. "Intelligent Agents". In Weiss, G. (eds.), Multiagent Systems, The MIT Press, 1999.

[31] M. Wooldridge, N.R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design", Autonomous Agents and Multi-Agent Systems, 3(3), Kluwer Academic Publishers, 2000, pp. 285-312.