The AgentComponent Approach, Combining Agents, and Components

Richard Krutisch, Philipp Meier, and Martin Wirsing

Ludwig-Maximilians-Universtität München
{krutisch, meierp, wirsing}@informatik.uni-muenchen.de

Abstract. In this paper we introduce a new approach, the so-called AgentComponent (AC) approach which combines component and agent technology. A multi agent system (MAS) is composed of AC instances, each AC instance consists of a knowledge base, storing the beliefs of an AC instance, of slots, storing the communication partners of an AC instance, of a set of ontologies, that represent domain specific languages for certain contexts, and of so-called ProcessComponents (PC) representing the behaviours of an AC instance. The AC is a generic component that can be reused (instantiated ACs) and parametrized by customizing the communication partners (slots), the ontologies and the behaviours (PCs) that can be added and removed from any AC instance. Hereby we achieve added value for agents and components. Agents can be easily composed, customized and reused whereas components get enhanced communication and interaction facilities from agents. We present this approach in detail, show how to construct a component-based MAS by a simple example and present a graphical tool for composing systems of AgentComponents.

1 Introduction

Software agents and software components are two emerging fields in software development. Each concept has its advantages and disadvantages but both concepts are improving the way large scaled software is built. While components focus more on reusability aspects of software, agents (MAS) focus on processing complex tasks as a community.

Agents. In agent software development we distinguish basic kinds of agent systems: Reactive systems, MAS (interaction-driven systems) and Belief-Desire-Intention systems. Reactive systems define the simplest form of agent systems. Such systems can easily be described by state machines, that react on events by triggering certain actions. MAS focus mainly on the interaction aspects of a system of agents that collaboratively execute complex tasks. In the BDI model agents have beliefs and plans and can trigger different plans to reach certain defined goals. In this paper we focus on building MAS. And therefore we shortly want to name the properties of agents in such a system [14,13]. Agents must be able to react on direct events (reactivity), moreover agents must be able to react on indirect events (proactivity) such as changes in the environment

M. Schillo et al. (Eds.): MATES 2003, LNAI 2831, pp. 1–12, 2003.
© Springer-Verlag Berlin Heidelberg 2003

or changes of the plans of an agent. Moreover agents should execute their tasks autonomously (autonomy) i.e. there should not be too much interaction with the user during process/task execution. And of course agents must be able to interact and exchange data and information by using certain protocols and ontologies (interaction, communication, social behaviour).

Components. To find a common definition for *components* is quite a big problem [6]. And in this paper we only point out the properties of components we think are useful for this approach. In [4] is stated: *"Component (in code): A coherent package of software implementation that (a) can be independently developed and delivered, (b) has explicit and well-specified interfaces for the services it provides, (c) has explicit and well-specified interfaces for services it expects from others, and (d) can be composed with other components, perhaps customizing some of their properties, without modifying the components themselves."* Considering components, one of the most important aspects is the reusability of the component in different contexts. And to provide certain functionality even for contexts which were not intended to be contexts for this component. To provide reusability we need to specify properly the components' interfaces, so that we are able to compose and exchange components in a component based software system. The interior of a component can be considered as a blackbox that is invisible for the user. Whereas the interfaces of a component describe, the services/methods a component provides and the services/methods it needs. Moreover it does not matter who a component provides. As long as a component is well specified it can easily be composed with other components to a more complex system.

Combining Agents and Components. Many similarities can be found between agent and component technology [11]. But we are more interested in the complementary concepts of these technologies, because these are the concepts that make the profit of combining agents and components. We see the main purpose of agents in the communication ability, which gives us the possibility to process complex tasks by assigning single tasks to different agents. Instead components focus more on reusability and parametrisation/customization aspects for the deployment of a component in different contexts. In our opinion there exist two different concepts of combining agent and component technology. The first one, we call it "agentifying", sees component technology as starting point and tries to include agent properties into existing components. The second one, we call it "componentifying", considers agent technology as starting point and tries to add component features to existing agent technology. Our approach which is based upon the "componentifying concept" combines the main features of agents and components in the so-called AgentComponent (AC). In this way we achieve both: An AC has communication abilities (the agent as starting point) and that can be reused and parametrized (the included component features) for different contexts (see Fig. 1). With our approach we want to disburden agent software development. The component technology is a suitable instrument for this task by making certain agent concepts customizable and reusable. Here "reusable" means to have an AC that can be instantiated for every agent we need. And

"customizable" means to connect AC instances, add/remove certain ontologies and add/remove behaviours graphically.

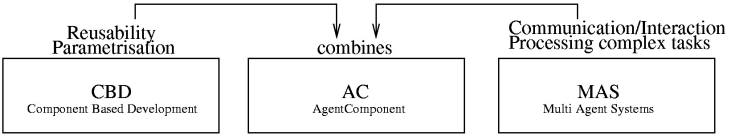


Fig. 1. Combination Profit

In section 2 we describe the basic concepts of the AC approach, where we give a simple example to illustrate these concepts. In section 3 we show how to construct AgentComponent Systems (ACS), i.e. systems composed of AgentComponents, and section 4 presents a graphical tool that helps to construct systems composed of AgentComponents. Finally, section 5 describes some related work and discusses differences to this approach, and section 6 concludes the paper.

2 AgentComponent

In general, the AC has all agent properties like autonomy, reactivity, proactivity and interaction and adds component features like reusability and customizability (according to the "componentifying concept"). An AC is a generic component that describes general services that every agent must provide and so an AC can be instantiated/reused for every agent one wants to build. A user can easily customize an AC instance for his purpose and for his context in the following ways: **(1)** Customizing the ACs' communication partners. We introduce so-called *slots* that hold information about communication partners of AC instances. Communication partners can be added to or removed from the slots of AC instances (customization of slots). **(2)** Customizing the ACs' ontologies. Ontologies can be registered or deregistered from the AC instance. **(3)** Customizing the ACs' processes. We introduce so-called *ProcessComponents* (PCs) that implement and describe simple or complex behaviours that can be added to or removed from any AC instance. With these three facilities the *appearance* and the *behaviour* of AC instances can be customized. Every AgentComponent includes the following entities for which it has to provide services. A knowledge base, slots (holding communication partners), a set of ontologies and a set of PCs:

Knowledge Base. The knowledge base is a basic element of every AC instance. An AC provides services to add, remove and get resources from the knowledge base. The following services are provided by an AC for the knowledge base concept:
(1) *retrieve* (getting knowledge from the knowledge base).

(2) *assert* (adding knowledge to the knowledge base).
(3) *retract* (removing knowledge from the knowledge base).

Ontologies. A well-known agent concept is the ontology concept that agents use to understand the content of the messages that are being sent or received. So for certain contexts ontologies must be provided that can be registered at AC instances. Instances of these ontologies can then be used as common language between AC instances and can also be asserted to, retracted or retrieved from the knowledge base. The following services are provided by the AC for the ontologies concept:
(1) *addOnto* (register an ontology at the AC instance).
(2) *removeOnto* (deregister an ontology from the AC instance).

Slots. Slots are entities of AC instances that hold information about communication partners (other agents). Communication pathways between AC instances can be created w.r.t. customized by connecting and changing slots, in this way filling the slots with the required destinations (other AC instances) for communication, and then messages can be sent and received via the created pathways. The following services are provided by the AC for the slot concept:
(1) *addSlot* (adds a slot for communication partners to AC instances).
(2) *removeSlot* (removes a slot from AC instances).
(3) *connectSlot* (fills a slot with destinations).
(4) *disconnectSlot* (removes a destination from a slot).

ProcessComponents. A PC represents a certain behaviour that can be added and removed to an AC instance.
(1) *addPC* (adds a PC to an AC instance).
(2) *removePC* (removes a PC from an AC instance).

When a PC has been added to an AC instance the PC has full use of all services that the AC provides (see Fig. 2). I.e. the PC is able to assert, retract, retrieve resources, to send and receive messages and moreover to use all the services listed in the StructuralInterface (see Fig. 2). So every PC implements an activity graph that consists of assert-, retract-, retrieve-, send-, receive-, StructuralInterface- or custom activities. In this way a PC defines which input it requires (retrieve, receive) and output it delivers (assert, retract, send, StructuralInterface services). By adding or removing PCs from an AC instance a system developer or user can determine the *appearance* (the structure of an AC instance, see StructuralInterface in Fig. 2) and the *behaviour* of an AC. That a PC determines a part of the behaviour of an AC instance is obvious, because a PC implements a certain workflow, represented by an activity graph. That a PC has the ability to determine the appearance of an AC instance is not so obvious and must be explained. We have mentioned before that a PC can use all the services that are included in the StructuralInterface. All these services define the *appearance* of an AC instance and are ment to customize the structure of an AC instance. On the one hand we want the user or system developer to do this structural customization graphically and on the other hand the PCs can implement behaviours that customize and determine the structure of an AC

instance. And therefore a PC has the ability to determine the *appearance* of an AC instance by using the StructuralInterface services.

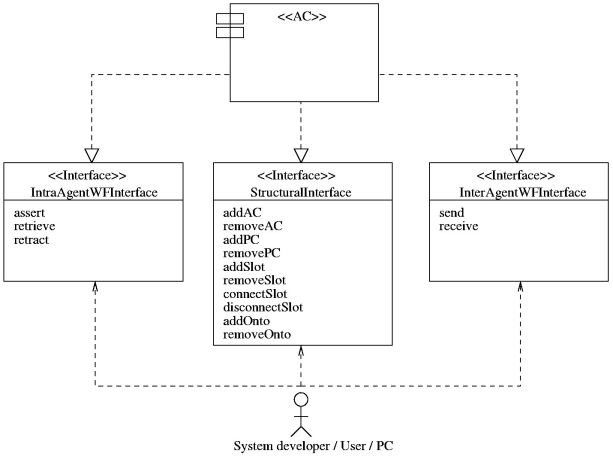


Fig. 2. AgentComponent Services

Intra-agent and Inter-agent Workflow. We have not yet talked about how AC instances or better their PCs can work together and define a workflow. This is important to explain, because we do not provide a workflow control within an AC and leave the workflow and its control to the PC implementor. The PCs themselves are supposed to work autonomously, i.e. PCs describe only their own workflow by defining what they require (receive, retrieve) and what they deliver (send, assert, retract, StructuralInterface services) not caring about the overall workflow. The way a workflow between two or more PCs is handled must be divided into two concepts, the inter-agent and the intra-agent workflow. Obviously the *inter-agent* workflow between AC instances is handled by sending and receiving messages using the pathways between AC instances. These pathways are stored in the slots and can be changed using the "connectSlot" or "disonnectSlot" service. Whereas the assert, retract and retrieve services handle the *intra-agent* workflow (workflow within an AC), by changing the knowledge base and therefore making resources available for other PCs (of the same AC instance) that wait for these resources. In this way we use the knowledge base as a "shared memory" for input and output variables, that can be used by all added PCs. The following services must be provided by an AC for the inter-agent workflow concept:
(1) *send* (sending messages to other AC instances).
(2) *receive* (receiving messages from other AC instances).

Putting all the mentioned concepts together the PC implementor has to provide:

(1) The implementation of the PC, where all services of the AC can be used to send, receive messages, to retrieve, assert and retract resources from the knowledge base and to customize the structure. **(2)** The PC description, where one must provide: (2.1) The description of all ontologies, that are required by the PC, (2.2) the information about required slots for the inter-agent communication and (2.3) the description of the workflow of the PC (mainly the description of the usage of the services), so that new PC providers understand the activities w.r.t. the behaviour of the PC.

Projects and AgentGroups. Agents are loosely coupled entities by nature, that communicate using messages and protocols and that are not directly aware of other agents around them (only indirectly via the AMS or the DF). We introduce the term *Project* as a special case of a PC. A Project normally describes an unspecified task, e.g. an AC instance does not know which other ACs are able to assist in collaboratively executing a task. So this AC has to look for one or more ACs that can handle this task. After this search they will join for this special PC at runtime and split up after finishing the Project. Projects are normally described by 1:n protocols and have a highly dynamic character. What misses in this highly dynamic environment is the fact that software systems only in some cases need such dynamic character. So, beside the Projects, we see the necessity to be able to connect the ACs directly using the before introduced slots. So we are able to build up an acquaintance net (AgentGroup) with an organization structure according to the pathways described in the slots. System developers or users will be able to create AgentGroups graphically using a tool, whereas Projects normally construct their organization structure and information pathways during runtime. The following services must be additionally provided by an AC for the AgentGroup concept:
(1) *addAC* (adds an AC to the AgentGroup of an existing AC instance).
(2) *removeAC* (removes an AC instance from an AgentGroup).

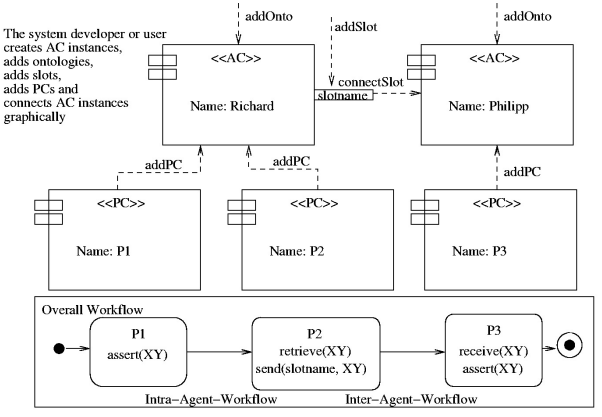


Fig. 3. Constructing a Simple AgentGroup

Example. To illustrate these basic concepts of the AC approach we describe a very small example. In this example (see Fig. 3) we describe the construction of a simple system, assuming that there exists a tool that supports the construction graphically. Furtheron, we assume that three PCs and their information description about ontologies, slots and behaviours are available. So we can use the three PCs called P1 to P3. P1 asserts an ontology object named XY to the knowledge base. P2 retrieves an XY ontology object and wants to send it to any AC instance available (we do not know to which AC instance we want to send so far, depends on the entries in the used slot). P3 receives a message containing an XY ontology object and asserts it to the ACs' knowledge base. P2 and P3 provide a description for the ontologies and for the slots they need during their workflow. First of all we create two idle AC instances named Richard and Philipp. Before we add any PC to Richard or Philipp we need to customize the structure of the both AC instances to make the intra-agent and inter-agent workflow of the PCs working. I.e. as we want to add P1 and P2 to Richard and P3 to Philipp, Richard and Philipp need the slots and the ontologies according to the PC descriptions. After creating a slot it can be filled with any destinations available, in our case we connect it with Philipp, and hereby Philipp is stored as a destination in a slot of Richard. By adding all this structural information and then adding the PCs as mentioned before we define on the one hand an intra-agent workflow where P2 requires the output (retrieve(XY)) of P1 (assert(XY)). And on the other hand we define an inter-agent workflow between P2 (send(XY)) added to Richard and P3 (receive(XY)) added to Philipp. After adding the PCs their behaviours start immediately and the workflow is executed (see Fig. 3).

Summary. We have shown an AC that provides three interfaces (InterAgentWorkflowInterface, IntraAgentWorkflowInterface and StructuralInterface) including common services every agent requires. System developers do not have to implement the AC, they just need to instantiate the AC as often as they require it. To fill the ACs with life PCs must be provided by the developers which then can be added to the AC instances. This results in an ACS (Agent Component System - assembled of ACs) which uses customized inter-agent communication (achieved by the concept of slots and ontologies) and uses customized intra-agent communication (achieved by the concept of PCs using the knowledge base).

3 Component-Oriented Development Process for AC-Systems

For constructing ACS (AgentComponent Systems) we propose a component-based software development process fully based on UML models [9]. We extend this process with the AC features and explain how we model these features. According to [9] we have four main phases in the AC development process:
(1) Requirements, (2) AC Identification, (3) AC Interaction and (4) AC Specification.

We concentrate in this paper on the construction phase of ACS that mainly takes place in the AC Identification and the AC Interaction phases. As a description of all phases would go beyond the scope of this paper we will not describe the AC Specification step, because it is mainly a refinement of identification and interaction using e.g. OCL (object constraint language) to model pre- and post-conditions of PCs. For the same reason we skip the requirements phase, though it is a very important phase, as it delivers the input, like business processes, business objects, actors and use case diagrams, that we need for the design phases (identification, interaction, specification).

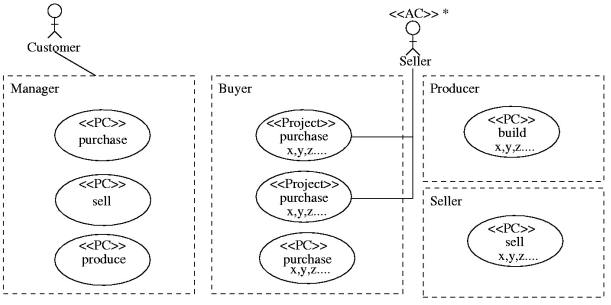


Fig. 4. Constructing AC Instances

AC Identification. Taking the input of the requirements phase we identify in this phase the agents (AC instances), roles and processes (PCs). We put all this information together to allocate the PCs to the AC instances and use UML-Use-Case diagrams [7,2] to model each AC instance with its PCs. These tailored Use-Case diagrams show AC instances and their PCs using stereotypes for AgentComponent («AC»), ProcessComponent («PC») and Project («Project»). In the example we have identified four AC instances (see Fig. 4) named Manager, Buyer, Producer and Seller and added different PCs to these AC instances. In the following we explain the Manager. In this small example we expect a Customer to order any Product at the Manager. The Manager itself has the Manager-like PCs added, that will initialize the Buyer to purchase all the parts, initialize the Producer to build any Product and initialze the Seller to sell the Product to the Customer.

Particularly the process of identifying the ACs in this example shows the way how ACS can be applied in practice. We start taking a found "real organization structure" (e.g. the organization structure of a certain company) from the requirements phase. ACS can map the "real organization structure" to the virtual organization structure where every organization element (unit,actor) is represented by an AC instance. In the example we have mapped a company structure consisting of four units Manager, Buyer, Producer and Seller to four AC instances with the same name.

AC Interaction. During this phase we describe the interaction between the AC instances we have found in the AC identification phase. We have following steps to do: (1) Modeling the organization structure, (2) defining the context ontologies and (3) describing the workflow of the identified PCs.

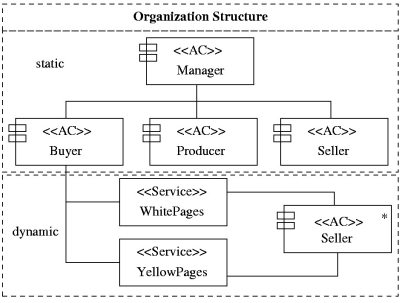


Fig. 5. Organization Structure

Modeling the organization structure. Here we describe the organization structure in an instantiated component model consisting of AC instances, yellow page services and white page services (see Fig. 5). This organization structure describes the acquaintance model (we call it AgentGroup) according to the pathways (stored in the slots) between the AC instances we have identified. Just to remember, we have identified the PCs before and therefore we have the PC description and the information which structural elements are required. For the organization structure we mainly need the slot information. The required slots must be created and filled with communication pathways (destinations) and that is what we model in this component diagram. If an AC instance (such as Buyer) includes a Project, that has dynamic behaviour and constructs its acquaintances during runtime, we model this by an association to the yellow or white pages (see Fig. 5).

Defining the context ontologies. In this step we define a context ontology for the AgentGroup. This ontology must describe all the concepts that the PCs need within their workflow. Fig. 6 provides an example ontology for purchasing, producing and selling cars modeled by a UML class diagrams.

Describing the workflow of the identified PCs. This step models the workflow of the identified PCs using UML activity diagrams [12]. In these diagrams we focus mainly on the WFInterface services *assert, retrieve, retract, receive and send* that the PCs use during their workflow. To disburden the activity diagrams we use an additional service named *change*, that models the change of an ontology instance in the knowledge base. We use one activity diagram for each PC. The ACs are represented by the swimlanes of the activity diagrams. Using these swimlanes we model the inter-agent workflow by sending objectflow states from one swimlane to another according to the created and

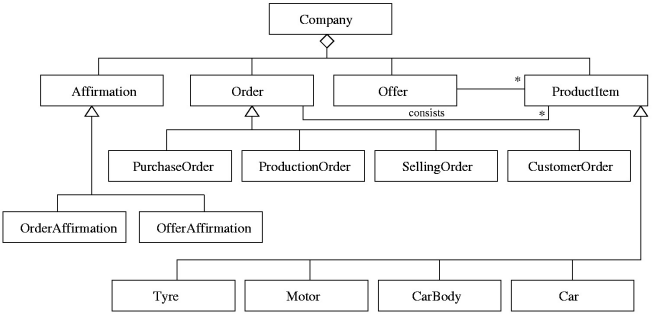


Fig. 6. Ontology example

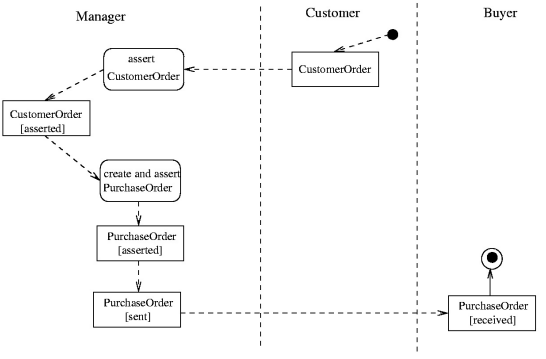


Fig. 7. Activity Diagram for PC "purchase" added to AC instance "Manager"

connected *slots*. The intra-agent communication is handled by the asserted, retracted, changed and retrieved statements. E.g. a PC waits (retrieve) until a special ontology object has been asserted (assert) by another PC of the same AC instance.

4 Tool Support

Currently we are developing the "AgentComponent Tool" (see Fig. 8) which allows one to build MAS graphically consisting of ACs as introduced in this paper (see section 2). This tool is based upon the JADE [8] framework for MAS and already provides following features:
(1) Distributed construction and runtime customization of ACS [15],
(2) cstomizing the structure of the AC instances by different users and PCs,
(3) setting customizing permissions for different users,
(4) creating non-ACs and storing them in the slots of AC instances,
(5) using the InterAgent- and IntraAgentWorkflow services during runtime via introspection.

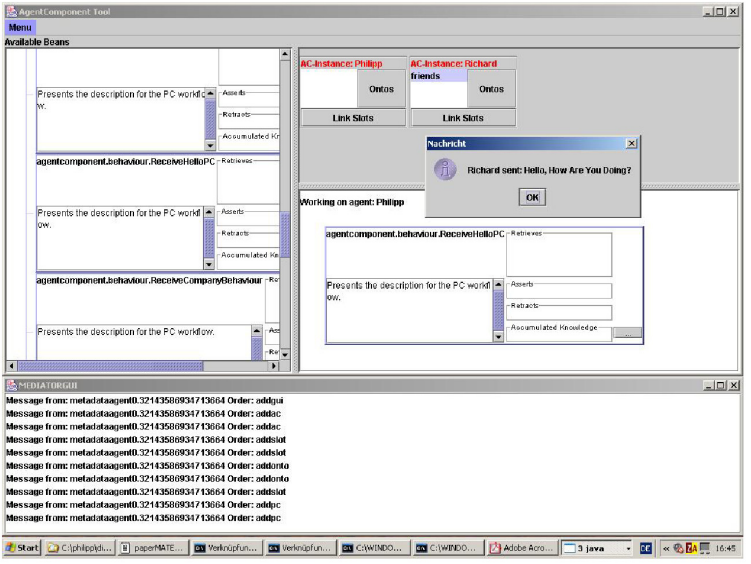


Fig. 8. AgentComponent Tool

5 Related Work

Although there exist many development and modeling approaches for agents [1, 3,10], the combination of agents and components is quite a new research field in agent technology. In our approach we borrow some concepts from existing works, e.g. the role model from [3], the activity diagram interaction protocols from [10, 12] (but in contrary to [10] we prefer a component- and model- oriented instead of a view-oriented development method) and different kinds of diagrams from the UML. And moreover we include combined agent and component technology aspects into these concepts. [5] also describes a component-based design method for agents. In contrast to [5] we have no compositional processes and knowledge, instead we have autonomously working processes (PCs). Moreover our AC provides infrastructure services but no workflow control for the processes as proposed by [5]. The AC approach does not focus on certain agent types like weak agents, strong agents or BDI-agents. The PCs can be implemented more or less *intelligently*, with beliefs, desires and goals or only with normal task execution functionality.

6 Concluding Remarks

Agent technology is a complex software technology. To make this technology easier to understand we need to encapsulate the complex concepts of agent technology in order to disburden the development of agent systems. And we found

the component technology as a suitable means to disburden agent software development by making certain agent concepts customizable (see section 2) and reusable (AgentComponent). For this purpose we have defined the AC as a component that provides certain services (the interfaces of the AC) for every agent. In this way we can instantiate and reuse the AC for every agent we want to build and instantiated ACs can be customized in the following ways: (1) customizing the structural elements of the AC instance (achieved by the services of the StructuralInterface), (2) customizing the inter-agent communication (achieved by the concept of slots) and (3) customizing the intra-agent communication (achieved by the concept of PCs using the knowledge base).

References

1. Agent UML. `http://www.auml.org`.
2. Bernd Oesterreich. *Objektorientierte Softwareentwicklung.* Oldenbourg, 1998.
3. David Kinny, Michael Wooldridge, Nicholas Jennings. The GAIA Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems 3*, pages 285 – 312, 2000.
4. Desmond Francis D'Souza, Alan Cameron Wills. *Objects, Components and Frameworks With UML.* Addison Wesley, 1999.
5. Frances M.T. Brazier, Catholijn Jonker, Jan Treur. Principles of Component-Based Design of Intelligent Agents. *Data and Knowledge Engineering*, 41:1 – 28, 1998.
6. Frank Griffel. *Componentware.* dpunkt.verlag, 1998.
7. Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison Wesley, 1998.
8. The JADE Programmers' Guide. `http://sharon.cselt.it/projects/jade/`.
9. John Cheesman, John Daniels. *UML Components, A Simple Process for Specifying Component-Based Softfware.* Addison Wesley, 2001.
10. Juergen Lind. The MASSIVE Development Method for Multiagent Systems. *In Proceedings of the Fifth International Conference on the Practical Application of Intelligent Agents and Multi-Agents (PAAM2000)*, 2000.
11. Juergen Lind. Relating Agent Technology and Component Models, 2001.
12. Juergen Lind. Specifying Agent Interaction Protocols with Standard UML. *In Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001)*, 2222, 2002.
13. M. Wooldridge and P.Ciancarini. Agent-Oriented Software Engineering: The State of the Art. *Agent-Oriented Software Engineering. Springer-Verlag Lecture Notes*, 1957, 2001.
14. Nick R. Jennings et al. Agent-Based Business Process Management. *ACM SIGMOD Record 27*, pages 32–39, 1998.
15. Philipp Meier, Martin Wirsing. Implementation Patterns for Visual Construction of Multi-Agent-Systems, Technical Report. `http://www.pst.informatik.uni-muenchen.de/publications/acpatterns.pdf`.