

UML Class Diagrams Revisited in the Context of Agent-Based Systems

Bernhard Bauer

Siemens AG, Corporate Technology, Information and Communications

Otto-Hahn-Ring 6
81739 Munich, Germany
(+49)89-636-50654

bernhard.bauer@mchp.siemens.de

ABSTRACT

Gaining wide acceptance for the use of agents in industry requires both relating it to the nearest antecedent technology (object-oriented software development) and using artifacts to support the development environment throughout the full system lifecycle. We address both of these requirements using AUML, the Agent UML (Unified Modeling Language) — a set of UML idioms and extensions. This paper illustrates the next steps of our approach by presenting notions for the internal behavior of an agent and its relation to the external behavior of an agent using and extending UML class diagrams.

General Terms

Algorithms, Documentation, Design, Reliability, Standardization, Languages, Theory.

Keywords

Agents, UML, internal behavior of agents, AUML, design artifacts, software engineering.

1. INTRODUCTION

Successful industrial deployment of agent technology requires techniques that reduce the risk inherent in any new technology. Two ways that reduce risk in the eyes of potential adopters are: to present the new technology as an incremental extension of known and trusted methods, and to provide explicit engineering tools that support industry-accepted methods of technology deployment.

We apply both of these risk-reduction insights to agents.

Accepted methods of industrial software development depend on standard representations for artifacts to support the analysis, specification, and design of agent software. Three characteristics of industrial software development require the disciplined development of artifacts throughout the software lifecycle. The scope of industrial software projects is much larger than typical academic research efforts, involving many more people across a longer period of time, and artifacts facilitate communication. The skills of developers are focused more on development methodology than on tracking the latest agent techniques, and artifacts can help codify best practice. The success criteria for industrial projects require traceability between initial requirements and the final deliverable — a task that artifacts directly support.

To leverage the acceptance of existing technology, we present

agents as an extension of active objects, exhibiting both dynamic autonomy (the ability to initiate action without external invocation) and deterministic autonomy (the ability to refuse or modify an external request). Thus, our basic definition of an agent is “an object that can say ‘go’ (dynamic autonomy) and ‘no’ (deterministic autonomy).” This approach leads us to focus on fairly fine-grained agents. More sophisticated capabilities can also be added, such as mobility, BDI mechanisms, and explicit modeling of other agents. Such capabilities are extensions to our basic agents, that is, they can be applied where needed, but are not diagnostic of agenthood.

The Unified Modeling Language (UML) is gaining wide acceptance for the representation of engineering artifacts in object-oriented software. Our view of agents as the next step beyond objects leads us to explore extensions to UML and idioms within UML to accommodate the distinctive requirements of agents. The result is Agent UML (AUML), see e.g. [1, 2]. This paper reports UML class diagrams revisited in the context of agent-based systems, namely the representation of the agent's internal behavior and relating it to the external behavior of an agent.

2. BACKGROUND

Agent UML (AUML) synthesizes a growing concern for agent-based software methodologies with the increasing acceptance of UML for object-oriented software development.

In [2] we have shown how Agent UML differs from the other existing agent software methodologies, as presented in [4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 26, 27].

This wide-ranging activity is a healthy sign that agent-based systems are having an increasing impact, since the demand for methodologies and artifacts reflects the growing commercial importance of our technology. Our objective is not to compete with any of these efforts, but rather to extend and apply a widely accepted modeling and representational formalism (UML) — one that harnesses insights and makes them useful for communicating across a wide range of research groups and development methodologies.

2.1 UML and AUML

To make sense of and unify various approaches on object oriented analysis and design, an Analysis and Design Task Force was established within the OMG. By November 1997, a de jure standard was adopted by the OMG members called the Unified Modeling Language (UML) [3, 17, 21]. UML unifies and

formalizes the methods of many approaches to the object-oriented software lifecycle, including Booch, Rumbaugh, Jacobson, and Odell.

In a previous paper, we have argued that UML provides an insufficient basis for modeling agents and agent-based systems [1, 2]. Basically, this is due to two reasons: *Firstly*, compared to objects, agents are active because they can take the initiative and have control over whether and how they process external requests. *Secondly*, agents do not only act in isolation but in cooperation or coordination with other agents. Multiagent systems are social communities of interdependent members that act individually.

To employ agent-based programming, a specification technique must support the whole software engineering process — from planning, through analysis and design, and finally to system construction, transition, and maintenance.

A proposal for a full life-cycle specification of agent-based system development is beyond the scope for this paper. Both FIPA and the OMG Agent Platform SIG are exploring and recommending extensions to UML. Moreover it is planned that within the European network of Excellence AgentLink a working group should be established on this topic. In this paper, we will focus on a new subset of an agent-based UML extension for the specification of the agent internal behavior of an agent and relating it to the external behavior of an agent using and extending UML class diagrams. This extension and considerations extends our effort on AUML for the software engineering process, because this topic closes the gap between the agent interaction protocol definition as shown e.g. in [2] and the internal behavior of an agent and its relation to the agent interaction protocols.

The definition of the internal behavior is part of the specification of the dynamical model of an agent system, as well as the static model of an agent.

3. UML CLASS DIAGRAMS - REVISITED

First of all let us have a closer look at the concepts of object oriented programming languages, namely the notions of object and class and adapt it afterwards to agent based systems.

3.1 Basics

In object oriented programming languages an object consists of a set of instance variables, also called attributes or fields, and its methods. Creating an object its object identity is determined. Instance variables are identifiers holding special values, depending on the programming languages these fields can be typed. Methods are operations, functions or procedures, which can act on the instance variables and other objects. The values of the fields can be either pre-defined basic data types or references to other objects.

A class describes a set of concrete objects, namely the instances of this class, with the same structure, i.e. same instance variables, and same behavior, i.e. same methods. There exists a standard method 'new', to create new instances of a class. A class definition consists of the declaration of the fields and the method implementations. It consists of a specification or an interface part as well as of an implementation part. The specification part describes, which methods with which functionality are supported by the class, but not how the operation is realized. The implementation part defines the implementation / realization of the methods and is usually not visible to the user of the method. The access rights define which methods are visible to the user and

which one are not. In most programming languages classes define also types, i.e. each class definition defines a type of the same name.

Some programming languages allow in class definitions also the definition of class variables, which are shared by all classes, in contrast to instance variables belonging to a single object. I.e. each instance of a class has its own storage for its instance variables, in contrast to class variables which share the same storage. Class variables are often used as a substitute for global variables. Beyond class variables, there are often used class methods which can be called independently of a created object and are used as global procedures.

3.2 Relating Objects with Agents

As already stated, an agent is more than an object, see figure 1.

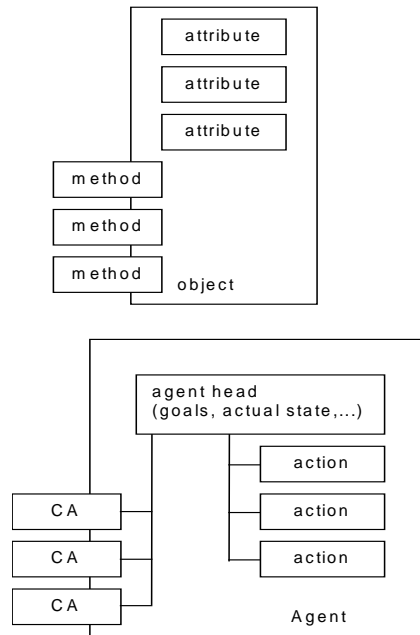


fig. 1: object vs. agent

We have autonomy, pro- and re-activity, the communication is based on speech act theory (communicative acts, CA for short), the internal state is more than only fields with imperative data types, and additional features. All these concepts have to be supported by a class diagram for agents.

In the agent oriented programming paradigm we have to distinguish between an *agent class* defining on the one side the type of an individual agent and being on the other side a blueprint for individual agents, and *individual agents*. I.e. an (individual) agent is an instance of an agent class. Therefore we specify the schema of an agent class which is then used in programs as instantiated agents.

An agent can be divided into the communicator - doing the physical communication, head - dealing with goals, states, etc. of an agent - and body - doing the pure actions of an agent. For the internal view of an agent we have to specify the agent's head and body.

The aim of the specification of the agent's internal behavior is to provide possibilities to define e.g. BDI semantics or permanent and actual goals as well as Java agents. Especially the semantics of the communicative acts and the reaction of an agent to some incoming messages have to be taken into consideration, this can be done either by the designer or the agent using e.g. BDI semantics. We allow the definition of a procedural as well as a declarative process description, the specification can e.g. be done using activity diagrams or the UML process specification language.

The reaction to events and pro-active behavior can be defined either by pro-active actions or agent head automata for pro-active behavior. Not only methods can be defined for an agent which are only visible to the agent itself, but actions which can be accessed by other agents. But in contrast to object orientation the agent decides itself whether some action is performed or not.

Abstract actions are characterized with pre-conditions, effects and invariants. Moreover the usual object oriented techniques have to be applied to agent technology, supporting efficient and structured program development, like inheritance, abstract agent types and agent interfaces, and generic agent types.

Single, multi, and dynamic inheritance can be applied for states, actions, methods, and message handling.

Associations are usable to describe e.g. agent A uses the services of agent B to perform a task (e.g. client, server), with some cardinality and roles. Aggregation and composition show e.g. car park service and car park monitoring can be part of an car park agent.

The components can either be agent classes or usual object oriented classes. Several times we have argued that agent and objects are completely different paradigms. Therefore we have to distinguish in our specifications between agents and objects. Especially an agent can be build using some object as part of its internal state (see fig 2). Therefore different notations between agents and objects have to be used either directly or using stereotypes.

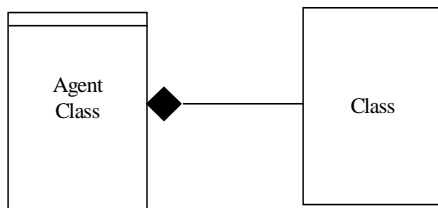


fig. 2. object part of an agent

Relating classes in the sense of objects to agent technology arises the question what is a class in the sense of an agents.

A class in the sense of object oriented programming is a blueprint for objects, in our context an agent class has to be a blueprint for agents. This can be either an instance of an agent or a set of agents satisfying some special role or behavior.

UML distinguishes different specification levels, namely the *conceptual*, the *specification* and the *implementation level*.

For the agent oriented point of view in the *conceptual level* an agent class corresponds to an agent role or agent classification, e.g. monitoring and route planning can be defined in different agent classes. E.g. we can have an individual traffic (IT) route planning (RP) agent and an IT Monitoring agent.

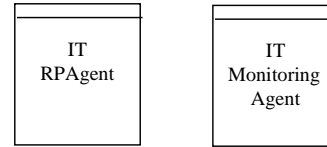


fig 3. conceptual level

On the *specification level* or *interface level* an agent class is blueprint for instances of agents, e.g. the monitoring and route planner are part of one agent class. But only the interfaces are described and not the implementation, i.e. the agent head automata (see below) describing the behavior of the agent according to incoming messages is missing. Only the internal states and the interface, i.e. the communicative acts supported by the agent, is defined.

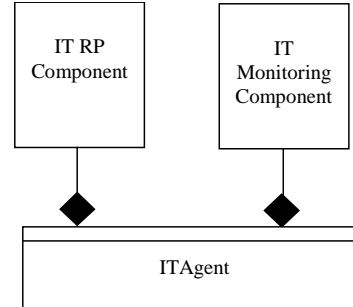


fig 4. specification level

The *implementational level* or *code level* is the most detailed description of a system, showing how instances of agents are working together and how the implementation of a class of agents look like. On this level the agent head automata has to be defined, too.

3.3 Agent class diagrams

In this section we show how usual UML class diagrams can be used and extended in the framework of agent oriented programming development. We will use the following notation to distinguish between different kinds of agent classes and instances.

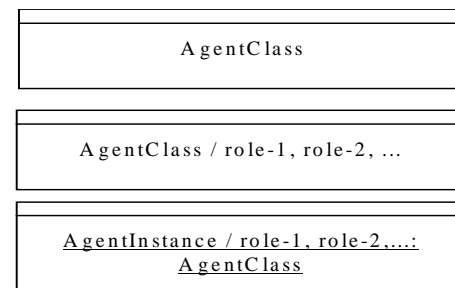
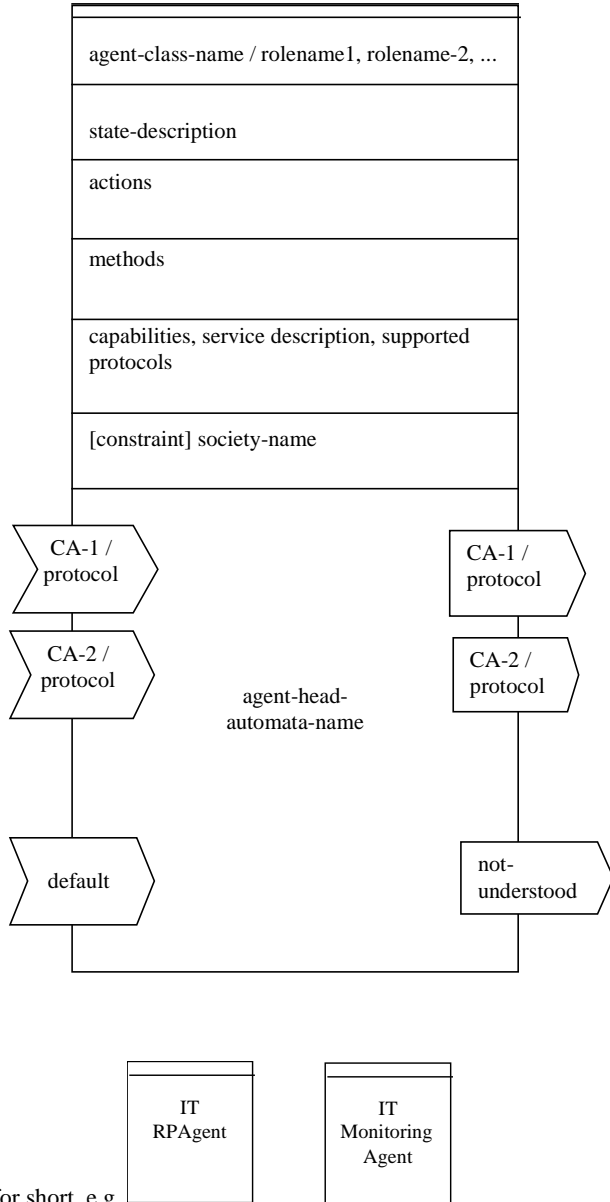


fig. 5: different kinds of agent classes

The first one denotes some agent class, the second some agent class satisfying distinguished roles and the last one defines some agent instance satisfying distinguished roles. The roles can be neglected for agent instances. According to the statement given above what has to be specified for agent classes we specify agents by the agent class diagram shown in figure 6



for short, e.g.

fig. 6. agent class diagram and its abbreviations

The usual UML notation can also be used to define such an agent class, but for more readable reasons we have introduced the above notation. Using stereotypes an agent class written as a class diagram can look as shown in fig. 7.

Agent Class Descriptions and Roles

In UML, *role* is an instance focused term. In the framework of agent oriented programming by *agent-role* a set of agents satisfying distinguished properties, interfaces, service descriptions or having a distinguished behavior are meant. UML distinguishes between multiple classification (e.g., a retailer agent acts as a buyer and a seller agent at the same time), and dynamic classification, where an agent can change its classification during its existence. Agents can perform various roles within e.g. one interaction protocol. In an auction between an airline and potential ticket buyers, the airline has the role of a seller and the participants have the role of buyers. But at the same time, a buyer in this auction can act as a seller in another auction. I.e., agents satisfying a distinguished role can support multiple classification and dynamic classification. Therefore, the implementation of an agent can satisfy different roles. An agent role describes two variations, which can apply within a multi agent system. A multi agent system can be defined at the level of concrete agent instances or for a set of agents satisfying a distinguished role and/or class. An agent satisfying a distinguished agent role and class is called agent of a given agent role and class, respectively. The general form of describing agent roles in agent UML (as we have shown in [2]) is

instance-1 ... instance-n / role-1 ... role-m : class

denoting a distinguished set of agent instances instance-1,..., instance-n satisfying the agent roles role-1,..., role-m with $n, m \geq 0$ and class it belongs to. Instances, roles or class can be omitted, in the case that the instances are omitted the roles and class are not underlined.

State description

The *state description* looks similar to a field description in class diagrams with the exception that we introduce a distinguished class *wff* for *well formed formula* for all kinds of logical descriptions of the state, independent of the underlying logic. With this extension we have the possibility to define as well BDI agents. Beyond the extension of the type for the fields, we allow in addition to the visibility attributes a persistency attribute which characterizes that the value of this attribute is persistence. E.g. in our personal travel assistance example the user agent can have an instance variable storing the already planned and booked travels. This field is persistent (denoted by the stereotype <<persistent>>) if the user agent is stopped and re-started later in a new session. Optionally the fields can be initialized with some values.

In the case of BDI semantics one can define four instance variables, named *beliefs*, *desires*, *intentions* and *goals* each of type *wff*. Describing the beliefs, desires, intentions and goals of a BDI agent. These fields can be initialized with the initial state of a BDI agent. The semantics states that the *wff* holds for the beliefs, desires, intentions and goals of the agent.

In a pure goal-oriented semantics two instance variables of type *wff* can be defined, named *permanent-goals* and *actual-goals*,

Capabilities

The capabilities of an agent can be defined either in an informal

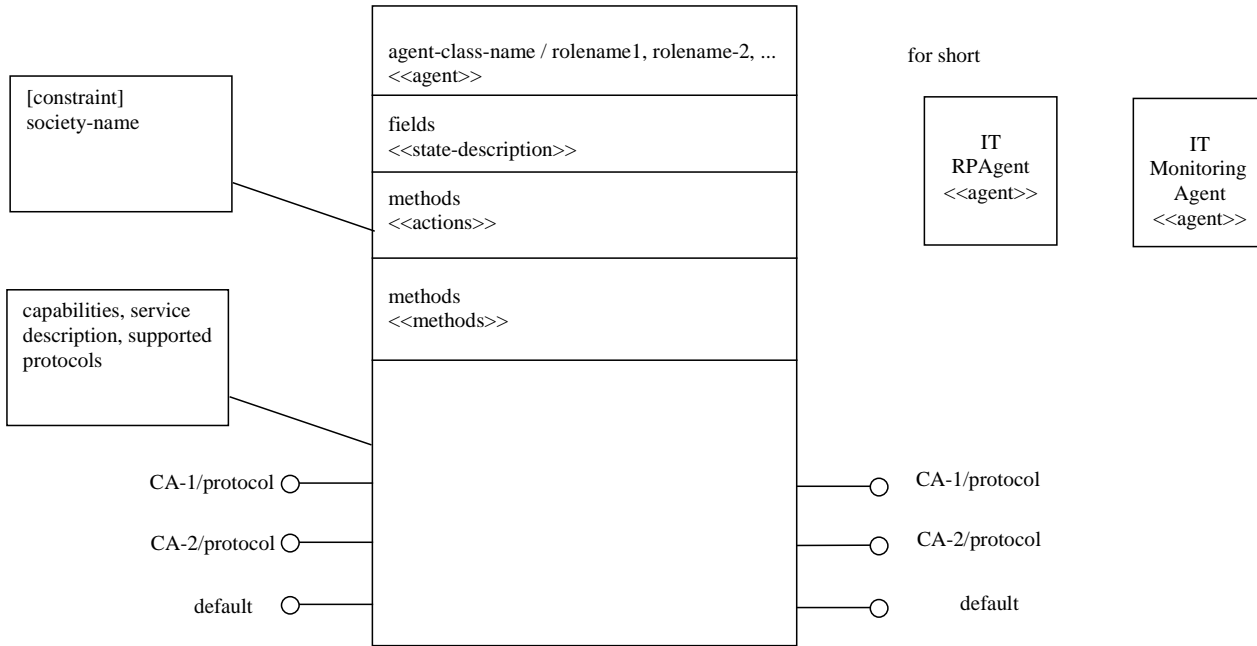


figure 7. using UML class diagrams to specify agent behavior and its abbreviations

holding the formula for the permanent and actual goals.

Usual UML fields can be defined if we have to specify a plain object oriented agent, i.e. an agent which is implementation on top of e.g. a Java-based agent platform, as e.g. JADE.

However in different design stages different kinds of agent can be appropriate, e.g. on the conceptual level one can specify some BDI agents which are then implemented by some Java-based agent platform, i.e. some refinement steps from BDI agents to Java agents are performed.

Actions

Pro-active behavior can be defined in two different ways, namely using pro-active actions and agent head automata with a pro-active behavior. The latter one will be considered later. Thus two kinds of actions can be specified for an agent: pro-active actions (denoted by the stereotype <<pro-active>>) are triggered by the agent itself, e.g. using timer, or a special state is reached. I.e. it is tested on state changes of the agent (e.g. timer, sensor input) if the pre-condition of the action evaluates to true. Re-active actions (denoted by the stereotype <<re-active>>) are triggered by another agent, i.e. receiving some message from another agent.

The description of an agent's actions consists of the action signature with visibility attribute, action-name and a list of parameters with its associated types. The semantics of an action is defined by pre-conditions, post-conditions, effects and invariants as in UML.

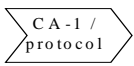

Methods

Methods are defined like in UML, eventually with pre-conditions, post-conditions, effects and invariants.

way or using class diagrams for e.g. FIPA-service descriptions

Sending and Receiving of Communicative Acts

The main interface of an agent to its environment is the sending and receiving of communicative acts. By communicative act (CA) we mean the type of the message as well as the other information, like sender, receiver or content like in FIPA-ACL messages. We assume that the information about communicative acts are represented by classes and objects. How ontologies and classes / objects are playing together is beyond this paper and are reason for future work.

The incoming messages are drawn as  and the outgoing messages are drawn as . The received or sent communicative act can either be some class or some concrete instance.

The notation *CA-1 / protocol* is used if the communicative act of class *CA-1* is received in the context of an interaction protocol *protocol*. In the case of an instance of a communicative act the notation *CA-1 / protocol* is used. As alternative notation we write *protocol[CA-1]* and *protocol[CA-1]*. The context */ protocol* can be omitted if the communicative act is interpreted independent of some protocol. In order to re-act to all kinds of received communicative acts, we use a distinguished communicative act *default*, which matches every incoming communicative act. The *not-understood* CA is sent if an incoming CA cannot be interpreted.

We distinguish between instances and classes, because of the following reasons:

An instance describes a concrete communicative act with fixed content or other fixed values. Thus if we have a concrete request, say "start auction for a special good", an instance of a communicative act would be used.

In order to allow a more flexible or generic description, say "start auction for any kind of good", of the interface of an agent classes are used for the communicative acts.

Matching of Communicative Acts

A received communicative act has to be matched against the incoming communicative acts of an agent to trigger the corresponding behavior of the agent. The matching of the communicative acts depends on the ordering of them, namely the ordering from top to bottom, since more than one communicative act of the agent can match an incoming message.

The simplest case is the default case, *default* matches everything and *not-understood* is the answer to messages not understood by an agent. Since we match on the one side instances of communicative acts, as well as classes of communicative acts, we have to define free variables within an instantiated communicative act. This is shown in figure 8 (class diagram for communicative acts where the instance variables have the type *undef*). Communicative acts are defined by classes without methods.

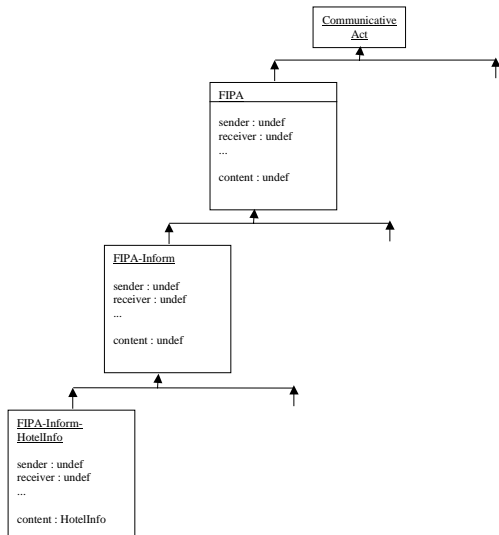


fig. 8 instance hierarchy on communicative acts, being an instance of the corresponding class hierarchy.

An input communicative act *CA* matches an incoming message *CA'*, iff

- *CA* is a class, then
 - *CA'* must be an instance of class *CA* or
 - *CA'* must be a subclass of class *CA* or a subclass of it.

- *CA* is instance of some class, then
 - *CA'* is instance of the same class as *CA* and
 - *CA.field* matches *CA'.field* for all fields *field* of the class *CA*, defined as
 - *CA.field* matches *CA'.field*, if *CA.field* has the value *undef*.
 - *CA.field* matches *CA'.field*, if *CA.field* is equal to *CA'.field* with *CA.field* not equal to *undef* and the type of *field* is a basic type.
 - *CA.field* matches *CA'.field*, if *CA.field* is unequal to *undef* and the type of *field* is not a basic data type and *CA.field* are instance of the same class *C* and *CA.field.cfield* matches *CA'.field.cfield* for all fields *cfield* of class *C*.

In the case of a communicative act in the context of a protocol, *protocol[CA]* matches *protocol'[CA']*, if *CA* matches *CA'* and *protocol'* is equal to *protocol*.

The analogous holds for outgoing messages, in this case the communicative act has to match the result communicative acts of the agent head automata.

3.4 Agent-Head-Automata

The agent head automata defines the behavior of an agent's head. We had defined an agent consisting of an agent's communicator, head and body.

The agent communicator is responsible for the physical communication of the agent.

The main functionality of the agent is implemented in the agent body. This can be e.g. an existing legacy software which is coupled to the MAS using wrapper mechanisms.

The agent's head is the "switch-gear" of the agent. Its behavior has to be specified with the agent head automata. Especially this automata relates the incoming messages with the internal state, actions and methods and the outgoing messages, called the reactive behavior of the agent. Moreover it defines the pro-active behavior of an agent, i.e. it automatically triggers different actions, methods and state-changes depending on the internal state of the agent. An example of a pro-active behavior is to do some action at a specific time, e.g. an agent migrates at pre-defined times from one machine to another one, or it is the result of some request-when communicative act.

UML supports four kinds of diagrams for the definition of dynamic behavior, namely sequence diagrams, collaboration diagrams on the object level, state and activity diagrams for other purposes. Sequence diagrams and collaboration diagrams are suitable for the definition of an agent's head behavior, since it is an object / agent instance focused diagram. So it can easily be used to define the concrete behavior, namely based on the actions, methods and state changes. It is up to the preferences of the designer to apply one of these diagrams. The state and the activity diagram is more suitable for a more abstract specification of the behavior of an agent's head. Again it is up to the designer to use one of these two diagrams.

Let us first of all have a closer look at the re-active behavior. We have to specify how the agent reacts to incoming. Using an extended state automata (see fig. 9) this behavior can be specified. In contrast to standard state automata the CA-notation of the class diagram is used to trigger an automata (initial states) and the final states match with the outgoing communicative acts.

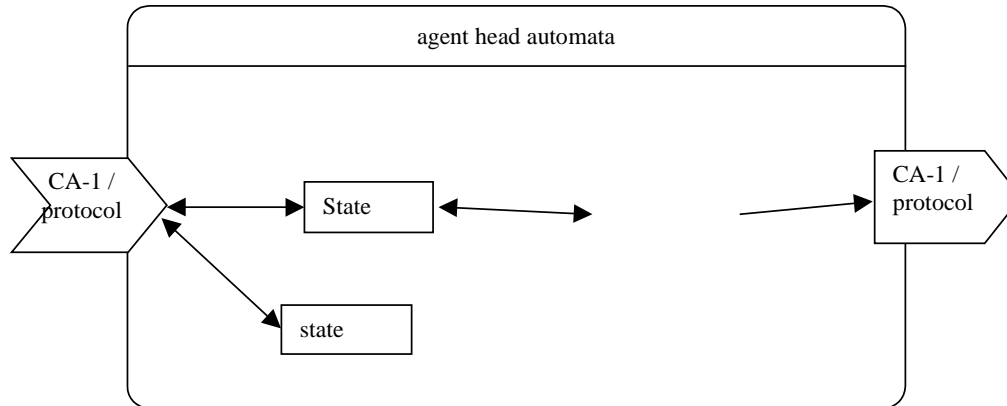


fig. 9 extended state automata

Pro-active behavior is not triggered by incoming messages, but depends on the validity of constraints or conditions. I.e. in the state automata the initial state(s) are marked with some conditions.

4. EVALUATION AND CONCLUSION

The artifacts for agent-oriented analysis and design were developed and evaluated in the German research project MOTIV-PTA (Personal Travel Assistant), aiming at providing an agent-based infrastructure for travel assistance in Germany (see www.motiv.de). MOTIV-PTA will run from 1996 to 2000. IT is a large-scale project involving approx. 10 industrial partners, including Siemens, BMW, IBM, DaimlerChrysler, debis, Opel, Bosch, and VW. The core of MOTIV-PTA is a multiagent system to wrap a variety of information services, ranging from multimodal route planning, traffic control information, parking space allocation, hotel reservation, ticket booking and purchasing, meeting scheduling, and entertainment.

From the end user's perspective, the goal is to provide a personal travel assistant, i.e., a software agent that uses information about the users' schedule and preferences in order to assist them in travel, including preparation as well as on-trip support. This requires providing ubiquitous access to assistant functions for the user, in the office, at home, and while on the trip, using PCs, notebooks, information terminals, PDAs, and mobile phones.

From developing PTA (and other projects with corporate partners within Siemens) the requirements for artifacts to support the analysis and design became clear, and the material described in this paper has been developed incrementally, driven by these requirements. So far no empirical tests have been carried out to evaluate the benefits of the Agent UML framework. However, from our project experience so far, we see two concrete advantages of these extensions: Firstly, they make it easier for users who are familiar with object-oriented software development but new to developing agent systems to understand what multi

agent systems are about, and to understand the principles of looking at a system as a society of agents rather than a distributed collection of objects. Secondly, our estimate is that the time spent for design can be reduced by a minor amount, which grows with the number of agent-based projects. However, we expect that as soon as components are provided to support the implementation

based on Agent UML specifications, this will widely enhance the benefit.

Areas of future research include aspects such as

- description of mobility, planning, learning, scenarios, agent societies, ontologies and knowledge
- development of patterns and frameworks
- support for different agent communication languages and content languages
- development of plug-ins for existing CASE-tools

At the moment we plan to extend the presented framework and take inheritance and the benefits and problems of inheritance into consideration.

5. REFERENCES

- [1] AUML: <http://www.auml.org>
- [2] Bauer, B.; Müller, J. P.; Odell, J.: An Extension of UML by Protocols for Multiagent Interaction, Proceeding, Fourth International Conference on Multi Agent Systems, ICMAS 2000, Boston, IEEE Computer Society, 2000.
- [3] Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Language User Guide*, Addison-Wesley, Reading, MA, 1999.
- [4] Brazier, F.M.T., Jonkers, C.M., Treur J., ed., *Principles of Compositional Multi-Agent System Development* Chapman and Hall, 1998.
- [5] Bryson, J., McGonigle, B. "Agent Architecture as Object Oriented Design," in: *Intelligent Agents IV: Agent Theories, Architectures, and Languages*. 1998.

- [6] Burmeister, B., ed., *Models and Methodology for Agent-Oriented Analysis and Design* 1996.
- [7] Burmeister, B., Haddadi A., Sundermeyer K., Generic, Configurable, Cooperation Protocols for Multi-Agent Systems, Lecture Notes in Computer Science, Vol. 957, 1995.
- [8] Garijo, F. J., Bomaned J., ed., *Multi-Agent System Engineering: Proceedings of MAAMAW'99*, 1999.
- [9] Gustavsson, R. E., "Multi Agent Systems as Open Societies," in: *Intelligent Agents IV: Agent Theories, Architectures, and Languages*, 1998.
- [10] Herlea, D. E., Jonker C. M., Treur J., and Wijngaards N.J.E., in: *Specification of Behavioural Requirements within Compositional Multi-Agent System Design*, 1999.
- [11] Iglesias, C. A., Garijo, M., González J.E., *A Survey of Agent-Oriented Methodologies*, in: *Intelligent Agents V: Agent Theories, Architectures and Languages* (ATAL-98), 1998.
- [12] Iglesias, C. A., Garijo, M., González, J. C., Velasco, J. R. "Analysis and Design of Multiagent Systems using MAS-CommonKADS," in: *Intelligent Agents IV: Agent Theories, Architectures, and Languages*, 1998.
- [13] Jonker, C. M., Treur, J., in: *Compositional Verification of Multi-Agent Systems: a Formal Analysis of Proactiveness and Reactiveness*, 1997.
- [14] Kinny, D., Georgeff, M., "Modelling and Design of Multi-Agent Systems," in: *Proceedings ATAL'96*, 1996.
- [15] Kinny, D., Georgeff, M., Rao, A., "A Methodology and Modelling Technique for Systems of BDI Agents," in: *MAAMAW'96*, 1996.
- [16] Lee, J., Durfee, E. H., "On Explicit Plan Languages for Coordinating Multiagent Plan Execution," in: *ATAL 98*, 1998.
- [17] Martin, J., Odell, J., *Object-Oriented Methods: A Foundation*, (UML edition), Prentice Hall, 1998.
- [18] Nodine, M. H., Unruh, A., "Facilitating Open Communication in Agent Systems: the InfoSleuth Infrastructure," *ATAL 98*, 1998.
- [19] Parunak, H. Van D., *Visualizing Agent Conversations: Using Enhanced Dooley Graphs for Agent Design and Analysis*, in: *Proceedings of the First International Conference on Multi-Agent Systems*, MIT Press, 1995.
- [20] Parunak, H. Van D., Odell J., *Engineering Artifacts for Multi-Agent Systems*, ERIM CEC, 1999.
- [21] Parunak, H. Van D., Sauter, J., Clark, S. J., *Toward the Specification and Design of Industrial Synthetic Ecosystems*, in: *ATAL 98*, 1998.
- [22] Rumbaugh, J., Jacobson, I., Booch G., *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.
- [23] Schoppers, M., Shapiro, D., *Designing Embedded Agents to Optimize End-User Objectives*, in: *ATAL 98*, 1998.
- [24] Singh, M. P., *A Customizable Coordination Service for Autonomous Agents*, in: *ATAL 98*, 1998.
- [25] Singh, M. P., *Towards a Formal Theory of Communication for Multi-agent Systems*, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pp. 69-74, Morgan Kaufmann, August 1991.
- [26] Wooldridge, M., Jennings, N. R., Kinny, D., "The Gaia Methodology for Agent-Oriented Analysis and Design," *International Journal of Autonomous Agents and Multi-Agent Systems*, 3, 2000.
- [27] Ciancarini, P., Wooldridge, M. J., eds, *Agent-Oriented Software Engineering*, First International Workshop, AOSE 2000, Limerick, Ireland, June 2000, 2001.