UML class diagrams revisited in the context of agent-based systems

Bernhard Bauer

Angaben zur Veröffentlichung / Publication details:

Bauer, Bernhard. 2002. “UML class diagrams revisited in the context of agent-based systems.” In *Lecture Notes in Computer Science*, 2222:101–18. Berlin [u.a.]: Springer. https://doi.org/10.1007/3-540-70657-7_7.

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
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.

1 Introduction

Agent technology enables the specification, design and implementation of future software systems characterized by situation awareness and intelligent behavior, distributedness, complexity as well as mobility support. Agent technology has the potential to play a key role in reaching goals of future applications and services like automating daily processes, supporting the (nomadic) user with pro-active and intelligent assistance providing adaptive and self-organizing system functionality, opening the way to new application domains while supporting the integration of existing and new software, and make the development process for such applications easier and more flexible. However successful industrial deployment of agent technology requires 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.

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 success criteria for industrial projects require traceability between initial requirements and the final deliverable — a task that artifacts directly support.

- The skills of developers are focused more on development methodology than on tracking the latest agent techniques, and artifacts can help codify best practice.

Agent technology is based on existing basic technologies like software technology, e.g. object-orientation, components, or plug-and-play technologies. The content is described using standards like XML (eXtensible Markup Language); existing communication mechanisms are applied within agent technology. But agent technology provides additional features to these technologies. On the one hand additional functionality is added, like matchmaking, agent mobility, cooperation and coordination facilities or adaptive preferences model. On the other hand the supported agent infrastructure consists of application frameworks that can be instantiated for special purposes. Moreover agent platforms allow an easy implementation of agent based applications and services. These pillars of agent technology result in personalized added value services, supporting the (nomadic) user with intelligent assistance, based on search, integration and presentation of distributed information and knowledge management, advanced process control support, and mobile & electronic commerce and enterprise applications.

Unfortunately the potential of agent technology from a software engineering point of view is not studied sufficiently to derive exact numbers about cost saving doing an agent oriented software development process. But having a look at the already performed projects we assume the usage of agent-oriented patterns, like communication protocols or agent architectures, can reduce the development and the risks inherent in any new technology.

At the moment no sufficient software processes and tools are available being well suited for industrial projects. 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] and the paper on MESSAGE and of Odell et al. in this volume). 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. In contrast to MESSAGE (see the paper on MESSAGE in this volume) with a focus on the analysis phase, our extensions support the design phase.

The rest of this paper is organized as follows. In section 2 the background on Agent UML is given. Afterwards UML class diagrams are revisited and extended in the framework of agents, having also a look at inheritance. The paper finishes with some evaluation and conclusions.

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.

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. Multi-agent 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 of this paper. Therefore we will focus on a new subset of an agent-based UML extension for the specification of the agent's internal behavior 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. Usually a standard method 'new' exists, to create new instances of a class. A class definition consists of the declaration of the fields and the method implementations, moreover of a specification or an interface part as well as of an implementation part.

The *specification part* describes the methods and their functionality supported by the class, but nothing is stated about the realization of the operation.

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 visibility of methods for specific users. In most programming languages classes define types, i.e. each class definition defines a type of the same name.

Some programming languages allow class variables within the definition of a class 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 sharing the same storage. Beyond class variables, class methods can be called independently of a created object. Both class variables and class methods can be used as global variables and global procedures, respectively.

3.2 Relating Objects with Agents

In contrast to an object that invokes its methods (see figure 1), an agent is able to evaluate incoming messages (communicative acts, CA for short) with respect to its goals, plans, tasks, preferences, and to the internal world model.

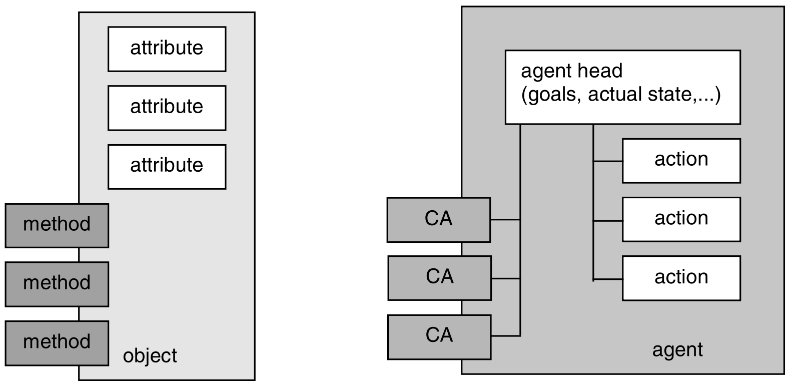


Fig. 1. Comparison object and agent

In contrast to distributed objects an agent is characterized by

- non-procedural behavior,
- balance between reactivity and pro-activity,
- high-level typed messages, like FIPA-ACL or KQML, and
- patterns and interaction protocols, not just client-server architecture.

In particular an agent is characterized by

- autonomy, i.e. deciding whether an action or behavior should be performed, not only method calls and returning results to other objects, especially not reacting based on hard-coded behavior
- pro- and re-activity, i.e. acting on events and messages as well as on self-triggered actions by the agents, in contrast to pure method invocation or events in the object oriented world
- the communication is based on speech act theory (communicative acts, CA for short), in contrast to method invocation with a fixed pre-defined functionality and behavior,
- the internal state is more than only fields with imperative data types, but include believes, goals and plans.

Among others these concepts have to be supported by a class diagram for agents.

3.3 Agent Class Diagrams

We start with some motivation and basics on agents, discuss afterwards the technical details, followed by agent head automaton and a look on generalization and inheritance for agents.

3.3.1 Motivation and Basics

An agent can be divided into

- the communicator - doing the physical communication,
- the head - dealing with goals, states, etc. of an agent and
- the body - doing the pure actions of an agent.

The communicator is responsible for the transparent dispatching of messages usually supported by agent platforms. For the internal view of an agent we have to specify the agent's head and body. The aim of the specification of an agent's internal behavior is to provide possibilities do define e.g. BDI, reactive, layered as well as interacting agent architectures and pure object oriented agents.

In particular the semantics of the communicative acts and the reaction of an agent to incoming messages have to be considered, established either by the designer of the multi-agent system or left open in the design phase and derived during run-time by the agent with reasoning functionality. The definition of a procedural as well as a declarative process description is supported using e.g. activity diagrams or the UML process specification language.

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 executed.

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 a 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 different things. 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.

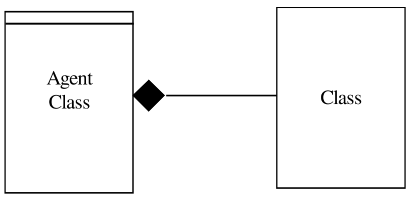


Fig. 2. Object part of an agent

3.3.2 Technical Details

In the agent oriented paradigm we have to distinguish between an *agent class*, defining a blue print for and the type of an individual agent, and *individual agents* being an instance of an agent class.

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 (for a detailed description see Odell et al. in this volume) 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.

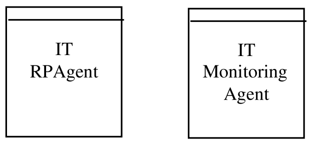


Fig. 3. Conceptual level

On the *specification level* or *interface level* an agent class is a 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 their implementation, i.e. the behavior of the agent to e.g. incoming messages is missing. Only the internal states and the interface, i.e. the communicative acts supported by the agent, are defined.

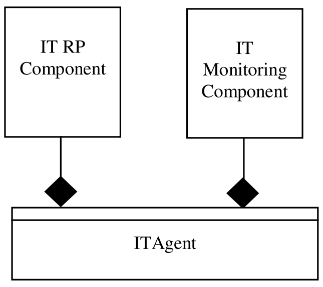


Fig. 4. Specification level

The *implementation level* or *code level* is the most detailed description of a system, showing how instances of agents are working together and how the implementation looks like. On this level the agent head automaton (see below) has to be defined, too.

In the following 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.

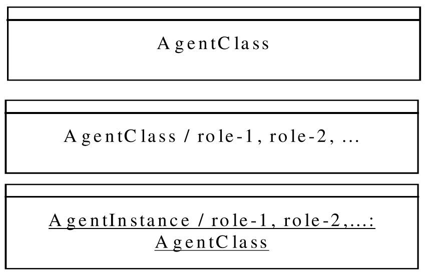

AgentClass
AgentClass / role-1, role-2, ...
AgentInstance / role-1, role-2,...:
AgentClass

Fig. 5. Different kinds of agent classes

The first one denotes an agent class; the second an agent class satisfying distinguished roles and the last one defines an agent instance satisfying distinguished roles. The roles can be neglected for agent instances. According to the above descriptions the agent class diagram shown in figure 6 specifies agent classes.

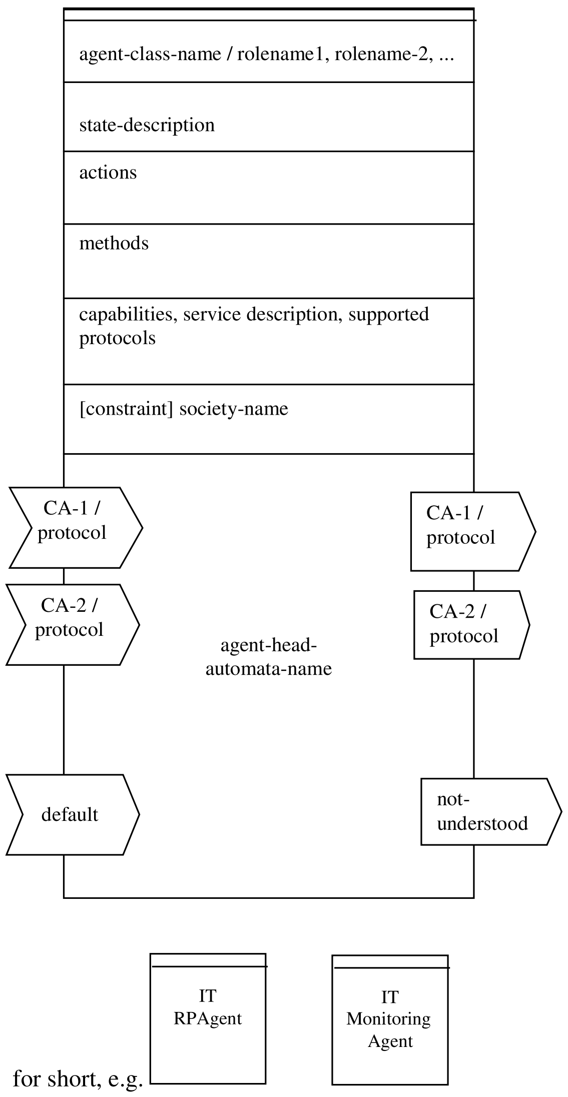


Fig. 6. Agent class diagram and its abbreviations

The usual UML notation can 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 is 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 is characterized. 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. It 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], for a detailed discussion on roles and agents see Odell et al. in this volume) 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 ≥ 0 and class it belongs to. Instances, roles or class can be omitted, for classes the role description is not underlined.

State Description

A *state description* is similar to a field description in class diagrams with the difference that a distinguished class *wff* for *well-formed formula* for all kinds of logical descriptions of the state is introduced, independent of the underlying logic. This extension allows the definition of e.g. BDI agents. Beyond the extension of the type for the fields, visibility and a persistency attributes can be added. E.g. in our personal travel assistance scenario the user agent has an instance variable storing the planned and booked travels. This field is persistent (denoted by the stereotype <<persistent>>) to allow the user agent to be stopped and re-started later in a new session. Optionally the fields can be initialized with some values.

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

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

Usual UML fields can be defined for the specification of a plain object oriented agent, i.e. an agent implemented on top of e.g. a Java-based agent platform.

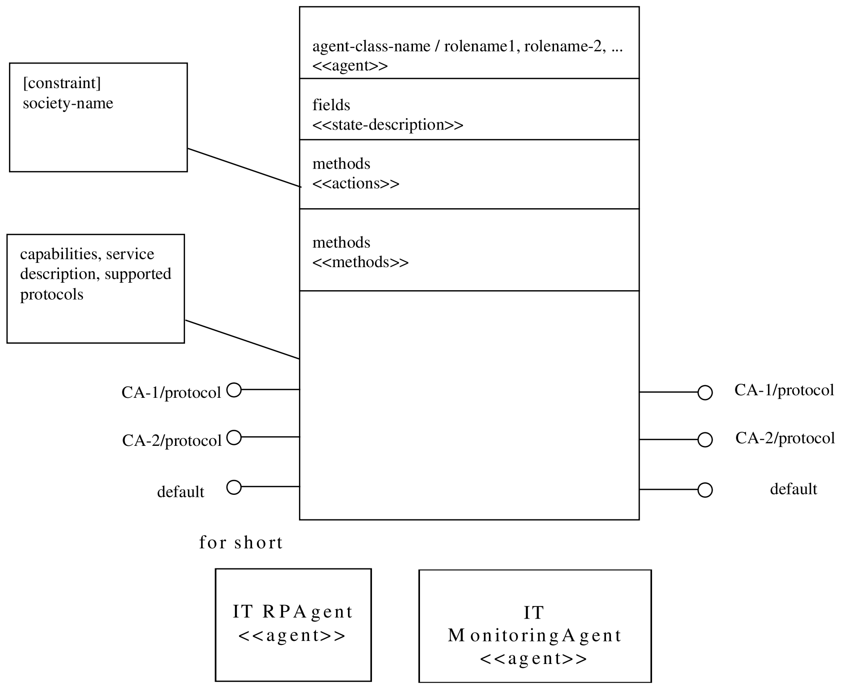


Fig. 7. Using UML class diagrams to specify agent behavior and its abbreviations

However in different design stages different kinds of agents can be appropriate, on the conceptual level BDI agents can be specified implemented by a Java-based agent platform, i.e. refinement steps from BDI agents to Java agents are performed during the agent development.

Actions

Pro-active behavior is defined in two ways, using pro-active actions and pro-active agent head automaton. 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, 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 a 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 associated types. Pre-conditions, post-conditions, effects, and invariants as in UML define the semantics of an action.

Methods

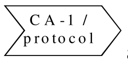
Methods are defined as in UML, eventually with pre-conditions, post-conditions, effects and invariants.

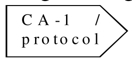
Capabilities

The capabilities of an agent can be defined either in an informal way or using class diagrams e.g. defining FIPA-service descriptions.

Sending and Receiving of Communicative Acts

Sending and receiving communicative acts characterize the main interface of an agent to its environment. By communicative act (CA) the type of the message as well as the other information, like sender, receiver or content in FIPA-ACL messages, is covered in this paper. We assume that classes and objects represent the information about communicative acts. 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 a class or a 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 applied. 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* matching any incoming communicative act. The *not-understood* CA is sent if an incoming CA cannot be interpreted.

Between instances and classes is distinguished, because an instance describes a concrete communicative act with a fixed content or other fixed values. Thus having a concrete request, say "start auction for a special good", an instance of a communicative act would be appropriate. To allow a more flexible or generic description, as "start auction for any kind of good", agent classes are used.

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, to deal with the case that more than one communicative act of the agent matches 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 instances of communicative acts are matched, as well as classes of communicative acts, free variables can occur within an instantiated communicative act, shown in figure 8 (class diagram for communicative acts where the instance variables have the type *undef*). Note, that classes without methods define communicative acts.

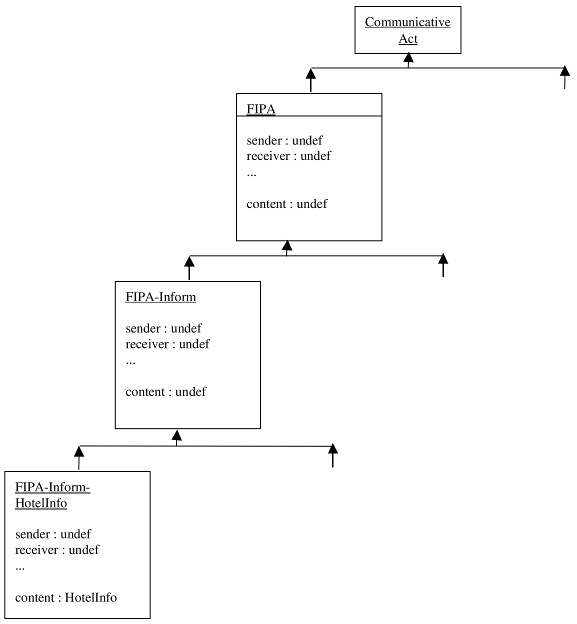


Fig. 8. Instance hierarchy on communicative acts, being an instance of the corresponding class hierarchy.

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 a 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, the communicative act has to match the result communicative acts of the agent head automaton.

3.4 Agent-Head-Automaton

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

The main functionality of the agent is implemented in the agent body, like existing legacy software coupled with the MAS using wrapper mechanisms.

The agent's head is the "switch-gear" of the agent. Its behavior is specified with the agent head automaton. In particular incoming messages are related to the internal state, actions and methods and the outgoing messages, called the *re-active behavior* of the agent. Moreover *pro-active behavior* of an agent, i.e. automatically triggering different actions, methods and state-changes depending on the internal state of the agent is specified. An example of a pro-active behavior is to do an action at a specific time point, e.g. an agent migrates at pre-defined times from one machine to another one, or it is the result of a request-when communicative act.

UML supports four kinds of diagrams for the definition of dynamic behavior: 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 instance-focused diagram. Thus the concrete behavior, based on actions, methods and state changes, can be easily defined. It is up to the preferences of the designer to apply one of these two diagrams. The state and activity diagram are more suitable for an abstract specification of an agent's behavior. Again it is up to the designer to select one of these two diagrams.

For the re-active behavior we specify how the agent reacts to incoming messages using an extended state automaton (see fig. 9). In contrast to standard state automaton the CA-notation of the class diagram is used to trigger an automaton (initial states) and the final states match with the outgoing communicative acts.

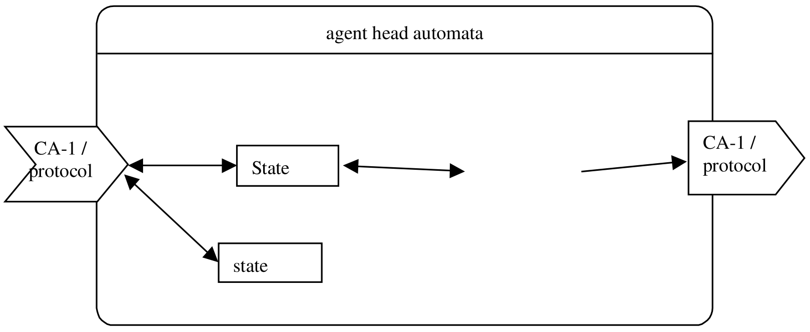


Fig. 9. Extended state automaton

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

3.5 Inheritance / Generalization on Class Diagrams

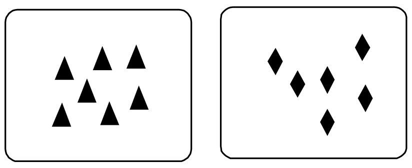
One main characteristic and THE modularization and structuring mechanism of object oriented programming languages is inheritance. This feature is applicable as well to agent-based systems.
Usually the following kinds of inheritance can be found in the literature:

- single inheritance,
- multiple inheritance and
- dynamic inheritance.

Reasons for introducing and using inheritance are

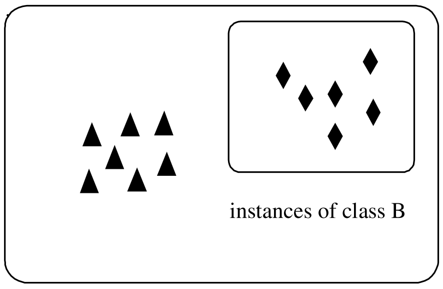
- the class hierarchy defines a type hierarchy
- inheritance supports re-usability and changeability
- inheritance supports to share common behavior.

Usually an object belongs either to a class A or to a class B, but it is impossible that an object of class A as well as to class B, see e.g. [28].



instances of class A instances of class B

But sometimes it is better to have a class to be an instance of two classes:



instances of class A

I.e. the instances of class B are also instances of class A. In agent's words, e.g. we have a buyer agent being in addition a seller agent. This inclusion property can be expressed using simple inheritance. Simple inheritance defines some type hierarchy, namely class B is some subtype of class A and the instances of class B are a subset of the instances of class A. If class B inherits from class A then B is subclass of A and A is super class of B.
Using single inheritance allows expressing, that an instance of class A and instances of class B can also be instances of class C.
For the implementation of the inheritance relation two views are well known:

- copy-view: inherits a class B from a class A, class B posses all components of class A, i.e. the instance variables and methods of class A. This is e.g. realized in the function tables of C++.
- search-view: An alternative view is obtained, if the access to instance variables and methods is done in the actual class. If the component is not present in this class, the super classes are searched for it. This technique is e.g. used in Smalltalk-80 and Java.

On the one side the concept of inheritance simplifies the implementation and on the other side makes the implementation more secure.

The implementation is simplified, since the code can be re-used, modified or extended using inheritance without re-writing all code. Methods can be re-defined, to specialize them for specific cases as well as define new methods to extend the functionality of a given class. Therefore the implementation is more secure, since tested code can be applied.

Thus inheritance supports top-down as well as bottom-up software development. Fixing the data type and their interfaces software systems can be developed top-down. Using class libraries and combining these classes with other classes results in a bottom-up software approach.

Generalization and specialization of class can be expressed as well using inheritance, since inheritance defines a type hierarchy, allowing an object to belong to more than one class. A generalization of classes is obtained, extracting common components of a class (instance variables and methods), which are used in different classes and defining them in an own class. This new class is a super class of the modified original class.

A specialization is realized, defining a new class B, which inherits from class A. Afterwards new components can be added in class B not included in class A.

A first draft characterization of simple inheritance for agents can look like

- subclass inherits all roles of the super class.
- state description:
  - additional fields can be defined, with new name and type
  - fields defined in a super class can be initialized
  - initialization can be re-defined, i.e. same type and field-name
- actions
  - are inherited from the super class
  - re-definition: same functionality, i.e. same argument types and result type as the super class; another new action is defined with different functionality, distinction between result types not possible.
- methods, like for actions but with possible result type and are not visible to other agents.
- capabilities are inherited , "everything" the super class can do, the actual class can do, too.
- societies
  - [constraint-1] society-name in super class and [constraint-2] society-name in subclass assumes that constraint-1 is stronger or weaker than constraint-2. Both cases are possible.
  - new societies can be added in the subclass

  - old ones are inherited and can be restricted or extended, see above.
- CAs
  - the search is performed like for method-calls in object-oriented programming languages, i.e. try to match the incoming CA with one of the input CAs starting with the first one from top to bottom. If no input CA matches the incoming CA than try to find a matching CA in the super class(es). If no CA matches with the exception of the "default" input CA, then the default input CA of the actual class is used, if defined otherwise the "default" input CA is recursive searched in the super class(es)
  - the output CAs are determined by the agent head automaton. Here the same holds, that all output CAs have to be defined in the class hierarchy.
- agent head automaton
  - are inherited from the super class
  - can be redefined with the same name
  - additional behavior to the agent can be added

This is a first draft characterization, but show that inheritance is also a very interesting aspect in the design of multi-agent systems, but a topic for future research.

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 run from 1996 to 2000. It was a large-scale project involving approximately 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.

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 Pro-activeness 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, Irland, June 2000, 2001.
28. A. Goldberg, D. Robson: Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, MA, 1983