UML Class Diagrams and Agent-Based Systems

Bernhard Bauer
Siemens AG, Corporate Technology, Information and Communications
Otto-Hahn-Ring 6, D-81370 Munich, Germany
(+49) 89 636 50654
bernhard.bauer@mchp.siemens.de

ABSTRACT
This paper illustrates the next steps of AUML by presenting a (A)UML presentation of the internal behavior of an agent and relating it to the external behavior of an agent using and extending UML class diagrams.

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.

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 [3] and idioms within UML to accommodate the distinctive requirements of agents. The result is Agent UML (AUML), see [1, 2, 4]. This paper reports on the representation of the agent's internal behavior and relating it to the external behavior of agent using and extending UML class diagrams.

2. UML CLASS DIAGRAMS - REVISITED
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, message handling. Associations are usable to describe e.g. agent A uses the services of agent B to perform a task, 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. Agent and objects are completely different paradigms. Therefore different notations between agents and objects have to be used either directly or using stereotypes. A class in the sense of object oriented programming is a blueprint for objects, 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.

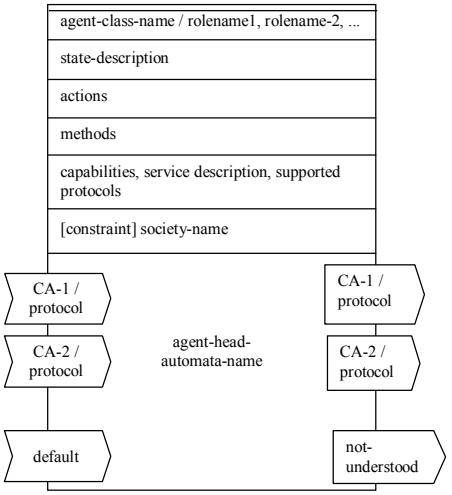


fig. 1. agent class diagram and its abbreviations

What has to be specified for agent classes is shown in figure 1.

Agent Class Descriptions and Roles
In UML, *role* is an instance focused term. In the framework of agent oriented programming by *agent-role* [1] a set of agents satisfying distinguished properties, interfaces, service descriptions or having a distinguished behavior are meant. Agents can perform various roles e.g. a reseller agent can act as a buyer and seller. 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. The general form (see [2]) of describing agent roles in Agent UML 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.

State description
The *state description* looks similar to a field description in class diagrams with the exception that a distinguished class *wff* for *well formed formula* for all kinds of logical descriptions of the state are used, independent of the underlying logic. In the case of BDI

https://doi.org/10.1145/375735.376010

semantics four instance variables can be defined, e.g. 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.

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

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 (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.

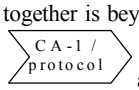
Methods

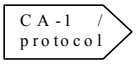
Methods are defined like in UML [2].

Capabilities

The capabilities of an agent can be defined either in an informal 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. 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. The context */ protocol* can be omitted if it 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.

An instance describes a concrete communicative act with fixed content or other fixed values, like a concrete request, say "start auction for a special good." In order to allow a more flexible or generic description, like "start auction for any kind of good," an agent class is 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 from top to bottom.

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. Communicative acts are defined by classes without methods.

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, *CA / protocol* matches *CA' / protocol'*, 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.

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 re-active 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 (for more details see [2]).

3. REFERENCES

[1] [Bauer, B.; Müller, J. P.; Odell, J.: *An Extension of UML by Protocols for Multiagent Interaction*, Proc. ICMAS 2000, Boston, 2000.

[2] Bauer, B., Müller, J.P., Odell, J.: *Agent UML: A Formalism for Specifying Multiagent Software Systems*, to be published International Journal on Software Engineering and Knowledge Engineering, 2001

[3] Martin, J., Odell, J., *Object-Oriented Methods: A Foundation*, (UML edition), Prentice Hall, 1998.

[4] www.auml.org