WHITESTEIN
Technologies

Software Development Methodology Solutions

Agent Modeling Language

Language Specification

Version 0.9

2004-12-20

Whitestein Technologies AG | Poststrasse 22 | CH-6300 Zug
Tel +41 41-711-1566 | Fax +41 41 711-1560 | http://www.whitestein.com

About this Document

Overview and Purpose

The purpose of this document is to provide the specification of the *Agent Modeling Language (AML)* metamodel, i.e.:

- ❒ to explain and precisely specify all the AML modeling elements/concepts in terms of their:
  - abstract syntax,
  - semantics,
  - notation, and
- ❒ to give guidelines and examples of their usage.

Outside the scope of the document is to give a "universal" glossary of precise definitions of *Multi-Agent Systems* (*MAS*) concepts. The reason for this is that in many cases there are no unique and generally agreed definitions. The document therefore defines only the semantics of AML modeling elements in conformance with the most general and commonly accepted understanding of MAS concepts. We hope that such an approach is flexible enough to accommodate all the specific nuances of MAS concept definitions and allows users to use their own concepts within the AML framework.

List of Authors

Radovan Cervenka, Whitestein Technologies
rce@whitestein.com

Ivan Trencansky, Whitestein Technologies
itr@whitestein.com

Table of Contents

1 Introduction

Purpose of AML

The ***Agent Modeling Language*** (***AML***) is a semi-formal[1] visual modeling language for specifying, modeling and documenting systems that incorporate concepts drawn from *Multi-Agent Systems* (*MAS*) theory.

The primary application context of AML is to systems explicitly designed using software multi-agent system concepts. AML can however also be applied to other domains such as business systems, social systems, robotics, etc. In general, AML can be used whenever it is suitable or useful to build models that:

- ❒ consist of a number of autonomous, concurrent and/or asynchronous (possibly proactive) entities,
- ❒ comprise entities that are able to observe and/or interact with their environment,
- ❒ make use of complex interactions and aggregated services,
- ❒ employ social structures,
- ❒ capture mental characteristics of systems and/or their parts, etc.

Motivation and goals

The most significant motivation driving the development of AML stems from the extant need for a ready-to-use, complete and highly expressive modeling language suitable for the development of commercial software solutions based on multi-agent technologies. Currently available MAS oriented modeling languages do not offer such facilities, because they are often:

- ❒ insufficiently documented and/or specified,
- ❒ using proprietary and/or non-intuitive modeling constructs,
- ❒ aimed at modeling only a limited set of MAS aspects,
- ❒ applicable only to a specific theory, application domain, MAS architecture, or technology,
- ❒ mutually incompatible, or
- ❒ insufficiently supported by *Computer-Aided Software Engineering* (*CASE*) tools.

Therefore AML is intended to be a language which overcomes the above mentioned deficiencies of the current state-of-the-art and practice in the area of MAS modeling languages. To qualify this more precisely, AML is intended to be a language that:

- ❒ is built on proven technical foundations,

[1] The term “semi-formal” implies that the language offers the means to specify systems using a combination of natural language, graphical notation, and formal language specification. It is not based on a strict formal (e.g. mathematical) theory.

- ❒ integrates best practices from *Agent-Oriented Software Engineering* (*AOSE*) and *Object-Oriented Software Engineering* (*OOSE*) domains,
- ❒ is well specified and documented,
- ❒ is internally consistent from the conceptual, semantic and syntactic perspectives,
- ❒ is versatile and easy to extend,
- ❒ is independent of any particular theory, software development process or implementation environment, and
- ❒ is supported by CASE tools.

The scope of AML

AML is designed to support business modeling, requirements specification, analysis, and design of software systems that use MAS concepts and principles.

The current version of AML offers:

- ❒ Support for the human mental process of requirements specification and analysis of complex problems/systems, particularly:
  - mental aspects, which can be used for modeling intentionality in business and use case models, goal-based requirements, problem decomposition, etc. (see section *6 Mental,* p. 137), and
  - contexts, which can be used for situation-based modeling (see section *7 Contexts,* p. 171).
- ❒ Support for the abstraction of architectural and behavioral concepts associated with multi-agent systems:
  - ontologies (see section *4.7 Ontologies,* p. 55),
  - MAS entities (see sections *4.2 Agents,* p. 25 , *4.3 Resources,* p. 27, and *4.4 Environments,* p. 28),
  - social aspects (see section *4.5 Social Aspects,* p. 30),
  - behavior abstraction and decomposition (see sections *5.1 Basic Behaviors,* p. 60, and *5.2 Behavior Decomposition,* p. 65),
  - communicative interactions (see section *5.3 Communicative Interactions,* p. 66),
  - services (see section *5.4 Services,* p. 101),
  - observations and effecting interactions (see section *5.5 Observations and Effecting Interactions,* p. 116),
  - mental aspects used for modeling mental attitudes of entities (see section *6 Mental,* p. 137),
  - MAS deployment (see section *4.6 MAS Deployment,* p. 48), and
  - agent mobility (see section *5.6 Mobility,* p. 128).

Outside the scope of AML

AML does not cover most operational semantics, which is often dependent on a specific execution model given by an applied theory or deployment

environment (e.g. agent platforms, reasoning engines, other technologies used).

The AML approach

Toward achieving the stated goals and overcoming the deficiencies associated with many existing approaches, AML has been designed as a language, which:

1. incorporates and unifies the most significant concepts from the broadest possible set of existing multi-agent theories and abstract models, modeling and specification languages, methodologies, agent platforms and multi-agent driven applications,
2. extends the above with new modeling concepts to account for aspects of multi-agent systems thus far covered insufficiently, inappropriately or not at all,
3. assembles these concepts into a consistent framework specified by the AML meta-model (covering abstract syntax and semantics of the language) and notation (covering the concrete syntax), and
4. is specified as a *conservative extension of UML*[2] to the maximum possible extent.

AML sources

Tab. 1-1 provides a summary of the most significant AML sources and their contribution to the language.

Source	***Used for***
UML 1.5 [52], UML 2.0 [53]	the underlying foundation of AML, its language definition principles (metamodel, semantics and notation), and extension mechanisms
OCL 2.0 [51]	the constraints language used in AML specification
MESSAGE [21], [44]	inspiration for the core AML modeling principles, partially also notation
AUML [3], [4], [5], [6], [46], [47], [48], [58]	inspiration for modeling of interactions, mobility and partially class diagrams
i* [1], [79],[80], GRL [28], [42], TROPOS [8], [68], KAOS [16], [70], NFR [11], [45], GBRAM [2], etc.	inspiration for and principles of goal-based requirements and modeling of intentionality
Gaia [77], [81], Styx [10], PASSI [12], [13], Prometheus [56], [57] TAO [63], AOR [73], [74], CAMLE [62], OPM/MAS [66], etc.	inspiration for modeling concepts and principles
OWL [55], [64], DAML [15], [23], OIL [22], [23], [49], etc.	inspiration for the specification and modeling of ontologies

Tab. 1-1 AML sources

[2] A conservative extension of UML is a strict extension of UML which retains the standard UML semantics in unaltered form [69].

Source	*Used for*
FIPA standards [24]	principles of MAS architecture and communication
Web services [72], OWL-S [43]	inspiration for the specification of services
existing agent-oriented technologies, e.g. agent platforms	architectural principles
modal, deontic, temporal, dynamic, epistemic and BDI logics [71], [75]	formal approach to the specification of DAI systems, extensions of the constraint language
MAS theories and abstract models, e.g. DAI [75], BDI [60], SMART [20]	identification of problems to model and principles of formal system specification

Tab. 1-1 AML sources

UML 2.0 as a base

AML is based on the *UML 2.0 Superstructure* [53], augmenting it with several new modeling concepts appropriate for capturing the typical features of multi-agent systems.

The main advantages of this approach are:

- Reuse of well-defined, well-founded, and commonly used concepts of UML.
- Use of existing mechanisms for specifying and extending UML-based languages (metamodel extensions and UML profiles).
- Ease of incorporation into existing UML-based CASE tools.

Structure of AML

AML is defined at two distinct levels—*AML Metamodel and Notation* and *AML Profiles*. Fig. 1-1 depicts these two levels, their derivation from UML 2.0 and optional extensions based on UML 1.* and 2.0.

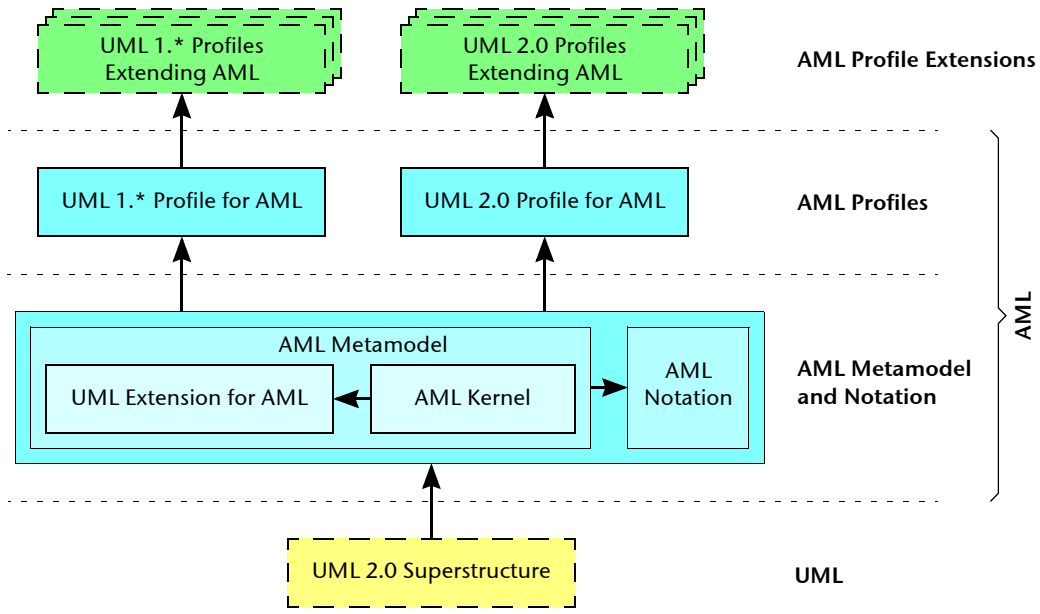


Fig. 1-1 Levels of AML specification

With reference to Fig. 1-1, the *UML* level contains the UML 2.0 Superstructure defining the abstract syntax, semantics and notation of UML. AML uses this level as the foundation upon which to define MAS-specific modeling constructs.

The *AML Metamodel and Notation* level defines the AML abstract syntax, semantics and notation. The *AML Metamodel* is further structured into two main packages: *AML Kernel* and *UML Extension for AML*.

The *AML Kernel* package is the core of AML where the AML specific modeling elements are defined. It is logically structured into several packages, each of which covers a specific aspect of MAS. The AML Kernel is a conservative extension of UML 2.0.

The *UML Extension for AML* package adds meta-properties and structural constraints to the standard UML elements. It is a non-conservative extension of UML, and thus is an optional part of the language. However, the extensions contained within are simple and can be easily implemented in most of the existing UML-based CASE tools.

At the level of *AML Profiles,* two UML profiles built upon the AML Metamodel and Notation are provided: *UML 1.* Profile for AML* (based on UML 1.*) and *UML 2.0 Profile for AML* (based on UML 2.0). These profiles, inter alia, enable implementation of AML within UML 1.* and UML 2.0 based CASE tools, respectively.

Based on AML Profiles, users are free to define their own language extensions to customize AML for specific modeling techniques, implementation environments, technologies, development processes, etc. The extensions can be defined as standard UML 1.* or 2.0 profiles. They are commonly referred to as *AML Profile Extensions*.

Extensibility of AML

AML is designed to encompass a broad set of relevant theories and modeling approaches, it being essentially impossible to cover all inclusively. In those cases where AML is insufficient, several mechanisms can be used to extend or customize AML as required.

Each of the following extension methods (and combinations thereof) can be used:

- ***Metamodel extension.*** This offers first-class extensibility—as defined by the *Meta Object Facility* (*MOF*) [50]—of the AML metamodel and notation.
- ***AML profile extension.*** This offers the possibility to adapt AML Profiles using constructs specific to a given domain, platform, or development method, without the need to modify the underlying AML metamodel.
- ***Concrete model extension.*** This offers the means to employ alternative MAS modeling approaches as complementary specifications to the AML model.

1.1 Target Audience

The document is intended to serve as a language specification and reference manual to AML users, software engineers, methodologists, and tool developers. Comprehension of the details and nuances of AML requires a deep understanding of UML 2.0 Superstructure [53]. The reader is assumed to have some prior knowledge of MAS concepts and AOSE terminology.

1.2 Specification Structure

Although the document is intended for advanced readers, we have tried to organize it in a way that it can be easily understood. In order to improve the readability and comprehension of the document, it is organized according to a hierarchy of packages which group either further (sub)packages or metaclasses that logically fit together. All the AML metaclasses are defined only within the packages on the lowest level of the package hierarchy, i.e. within packages that do not contain further subpackages. For details on the package structure of AML see section *3 Overall AML Package Structure* (p. 22).

Package specification

Each package specification section contains an Overview and either Abstract syntax (for the lowest level packages) or Package structure (for non-lowest level packages) subsections. The lowest level packages also contain class specification subsections for all the metaclasses defined within them. Package specification sections are structured as follows:

Overview (mandatory)
Natural language introduction to the content of the package.

Abstract syntax (mandatory for lowest level packages)
AML metamodel class diagrams (i.e. the metaclasses and their relationships) contained within the package.

Package structure (mandatory for non lowest level packages)
The package diagram of (sub)packages contained within the package together with their mutual dependencies.

Class specification

Each AML metaclass is specified in a separate subsection of a package section. Class specification sections contain all the mandatory and some (possibly zero) of the optional subsections described in what follows:

Semantics (mandatory)
A summary of semantics and an informal definition of the metaclass specifying the AML modeling element.

Attributes (optional)
All owned meta-attributes of the metaclass are specified in terms of:

- name,
- type,
- multiplicity,
- natural language explanation of semantics, and

- if applicable, descriptions of additional property values that apply to the meta-attribute (ordered, union, etc.).

The structure of the meta-attribute specification is as follows:

name: *type*[*multiplicity*]	*Semantics; description and specification of additional property values*

Associations (optional)
The owned (navigable) ends of meta-associations are described in the same way as the meta-attributes.

Constraints (optional)
Set of invariants for the metaclass, which must be satisfied by all instances of that metaclass for the model to be meaningful. The rules thus specify constraints over meta-attributes and meta-associations defined in the metamodel. Specification of how the derived meta-attributes and meta-associations are derived are also included. All the invariants are defined by the *Object Constraint Language* (*OCL*) [51] expressions accompanied with an informal (natural language) explanation.

Notation (mandatory for all but enumerations)
In this section the notation of the modeling element is presented.

Enumeration values (mandatory for enumerations)
The values of the enumeration metaclass are described in terms of:

- value,
- keyword, and
- semantics.

Presentation options (optional)
If applicable, alternative ways of depicting the modeling element are presented.

Style guidelines (optional)
An informal convention of how to present (a part of) a modeling element.

Examples (optional)
Examples of how the modeling element is to be used.

Rationale (mandatory)
A brief explanation of the reasons why a given metaclass is defined within AML.

1.3 Organization of the Document

The remainder of the document is structured as follows: *2 Extensions to Standard UML Notation* (p. 17) introduces and specifies several presentation options to some UML elements in order to provide a more intuitive and comprehensive notation. In section *3 Overall AML Package Structure* (p. 22), the overall package structure of the AML meta-model is presented. Sections

4 Architecture (p. 23), *5 Behaviors* (p. 60), *6 Mental* (p. 137), and *7 Contexts* (p. 171) contain the specification of all the packages of the AML Kernel (Architecture, Behaviors, Mental, and Contexts), their subpackages and metaclasses. In section *8 UML Extension for AML* (p. 174), the UML extension for AML is specified. Section *9 Diagrams* (p. 177) contains the specification of the AML-specific diagrams and diagram frames, section *10 AML as a UML Profile* (p. 181) the specifications of the AML profiles, and section *11 Extension of OCL* (p. 234) definition of AML extensions of OCL. Appendices are: *Appendix A References* (p. 236), *Appendix B AML Notation Summary* (p. 243), and *Appendix C Glossary* (p. 263).

1.4 Acknowledgements

The authors are indebted to the technical reviewers: Stefan Brantschen, Monique Calisti, and Dominic Greenwood, for their support and fruitful comments which have inspired many ideas and thus substantially influenced the current version of AML.

The authors would also like to acknowledge the contributions of and discussions with members of the MAS and AOSE communities.

1.5 Issue Reporting

AML specification is subject to continuous review and improvement. As part of this process we encourage readers to report and discuss issues on the AML mailing list or directly with the authors.

To subscribe to the AML mailing list, send an e-mail to:

sympa@lists.whitestein.com

with ‘subscribe aml-forum’ (without the quotes) in the subject or the message body. If you have problems subscribing, please send an e-mail to:

listmaster@whitestein.com

2 Extensions to Standard UML Notation

This section specifies presentation options for some UML elements in order to provide more intuitive and comprehensive notation. This alternative notation is then commonly used in following sections of the document to describe specific modeling elements of AML.

2.1 Stereotyped Classifier

The stereotype of a Classifier can, additionally to the stereotype keyword, be expressed also by an iconic notation. The possible notational variations are shown in the example of an AgentType (p. 25), Fig. 2-1.

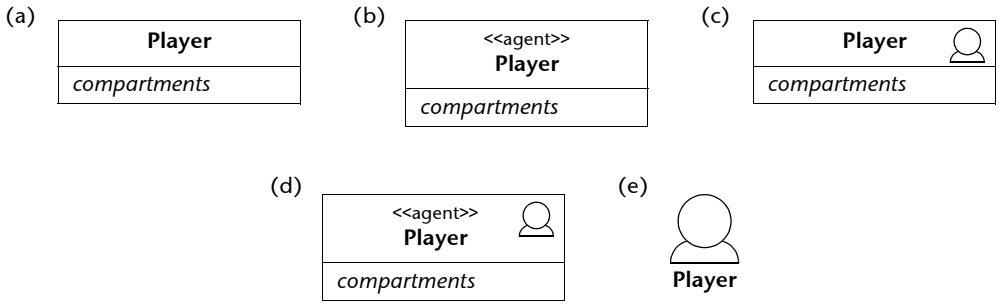


Fig. 2-1 Alternative notation for stereotyped Classifier. *The stereotype is: (a) hidden, (b) shown as a label, (c) shown as an iconic decoration, (d) shown as a label and iconic decoration, (e) shown as a large icon.*

2.2 ConnectableElement with a Stereotyped Type

The fundamental type[3] of a ConnectableElement's type can be visually expressed by its iconic notation, if defined. The possible notation variations are (see Fig. 2-2):

- a small stereotype icon shown in the upper right corner of the ConnectableElement's rectangle (so called iconic decoration), or
- a large stereotype icon used instead of ConnectableElement's rectangle.

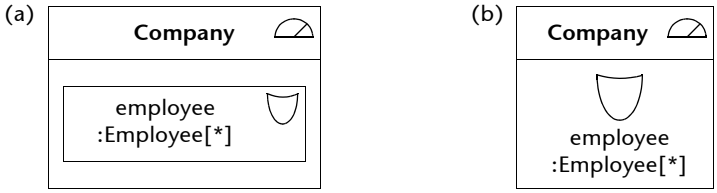


Fig. 2-2 Alternative notation for a ConnectableElement with stereotyped type. *The type's stereotype is shown as: (a) an iconic decoration, (b) a large icon.*

[3] By *fundamental type* of a modeling element we understand the metaclass of which this element is an instance.

The stereotype of the ConnectableElement itself (if defined) can be specified as a label or a small icon located in front of the the ConnectableElement's name.

Fig. 2-3 shows an example of an OrganizationUnitType (p. 31) Company, that consists of employees which are subordinates. The other OrganizationUnitType Bank consists of accountOwners which are peers to the Bank. Both parts, i.e. employees and accountOwners, are SocialProperties (p. 35) of the corresponding OrganizationUnitTypes with their own stereotypes <<sub>> (shown as a label) and <<peer>> (shown as a small half-black triangle icon) respectively.

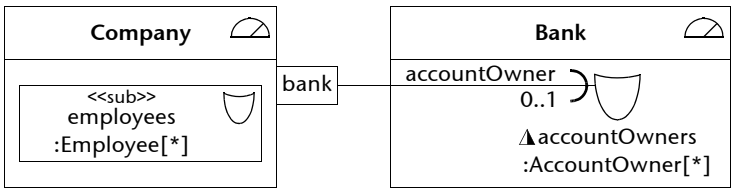


Fig. 2-3 Example of stereotyped parts having also stereotyped types

2.3 Connector with a Stereotyped Type

AML defines a few specialized Associations and Properties having own notation (see sections *4.5.5 SocialAssociation,* p. 38, *4.5.3 SocialProperty,* p. 35, etc.).

In addition to standard UML notation, AML allows the depiction of Connectors with the notation of the fundamental types of their types, and ConnectorEnds with the notation of the fundamental types of the referred Properties.

Fig. 2-4 shows an example of two Connectors (typed by SocialAssociations) and their ends (referring SocialProperties) depicted by iconic notation.

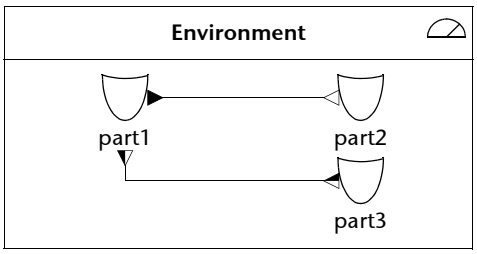


Fig. 2-4 Alternative notation for Connectors having SocialAssociation as a fundamental type of their types

2.4 Lifeline with a Stereotyped Type

To depict the fundamental type of the type of a ConnectableElement represented by a Lifeline, AML allows the placement of the stereotype small icon of that fundamental type into the Lifeline's "head", or to replace the Life-

line's "head" by the iconic notation of the fundamental type of the ConnectableElement's type. For an example see Fig. 2-5.

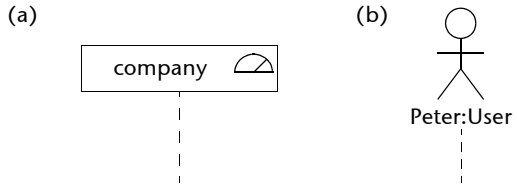


Fig. 2-5 Alternative notation for Lifeline. *The fundamental type of represented element is shown as: (a) an icon decoration, (b) a large icon.*

The same notation can be applied also for a Lifeline representing an inner ConnectableElement (see Fig. 2-6 for an example). The corresponding class diagram is shown in Fig. 4-8.

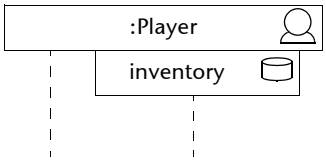


Fig. 2-6 Stereotype specified for the Lifeline representing an inner ConnectableElement

To differentiate the stereotype of the ConnectableElement represented by a Lifeline from the fundamental type of the ConnectableElement's type, the ConnectableElement's stereotype label is placed in the Lifeline's "head" or a small stereotype icon is placed in front of the ConnectableElement's name. In Fig. 2-7 an OrganizationUnitType (p. 31) Company contains RoleProperty (p. 41) employees, but it can also simultaneously play the entity role accountOwner. The corresponding class diagram is shown in Fig. 2-3.

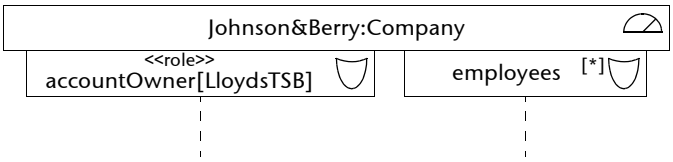


Fig. 2-7 Stereotype specified for the Lifeline representing an entity role

Extending the Lifeline's "head" by a stereotype of the fundamental type of the represented ConnectableElement's type can be applied also for Lifelines in Communication Diagrams. See example in Fig. 2-8.

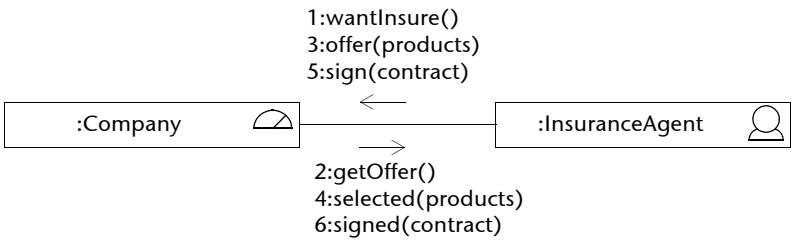


Fig. 2-8 Alternative notation for Lifeline in Communication Diagrams

2.5 Composed Lifelines in Communication Diagrams

Lifelines that represent owned attributes of a StructuredClassifier can be nested within the Lifeline representing their owner. Parts are depicted as rectangles with solid outlines, and properties specifying instances that are not owned by composition are shown as rectangles with dashed outlines. Ports can be depicted as well. See example in Fig. 2-9.

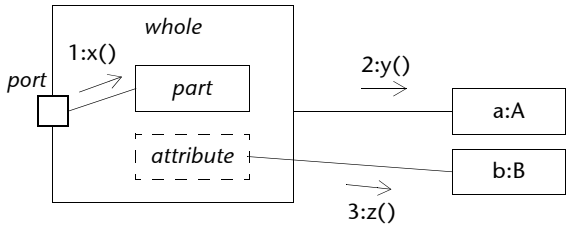


Fig. 2-9 Alternative notation for composite Lifelines in Communication Diagrams

2.6 ObjectNode with a Stereotyped Type

The presentation option described in the following text applies for all concrete subclasses of the ObjectNode metaclass.

The fundamental type of an ObjectNode's type can be visually expressed by its iconic notation, if defined. The possible notation variations are (see Fig. 2-10):

- a small stereotype icon placed in the upper right corner of an ObjectNode's rectangle shown in a standalone style, or
- a large stereotype icon used instead of a standalone ObjectNode's rectangle, or
- a small stereotype icon placed in the middle of a pin style ObjectNode rectangle.

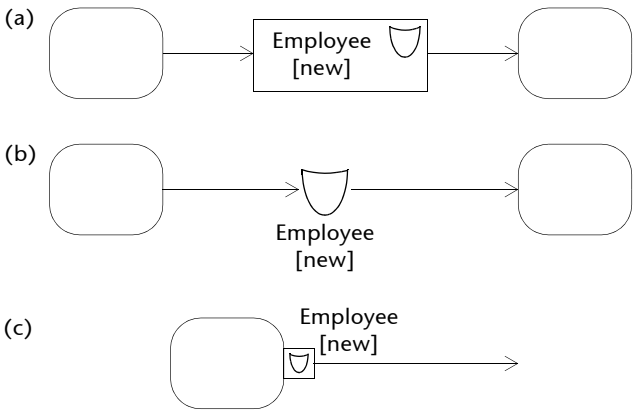


Fig. 2-10 Alternative notation for an ObjectNode with a stereotyped type. *The type's stereotype is shown as: (a) an iconic decoration in a standalone notation style, (b) a large icon in a standalone notation style, (c) a small icon in a pin notation style.*

The stereotype of the ObjectNode itself (if defined) can be specified as a label or a small icon located in front of the ObjectNode's name.

Fig. 2-11 shows a fragment of an Activity named EliminateIntruder, having an ActivityParameterNode intruder with its own stereotype <<byref>>, while the stereotype of its type is depicted as an iconic decoration.

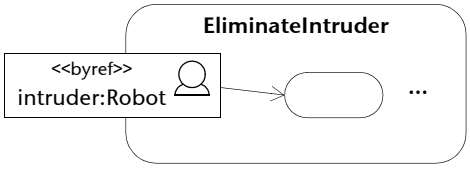


Fig. 2-11 Example of a stereotyped ObjectNode having also stereotyped type

3 Overall AML Package Structure

The overall package structure of the AML metamodel is depicted in Fig. 3-1.

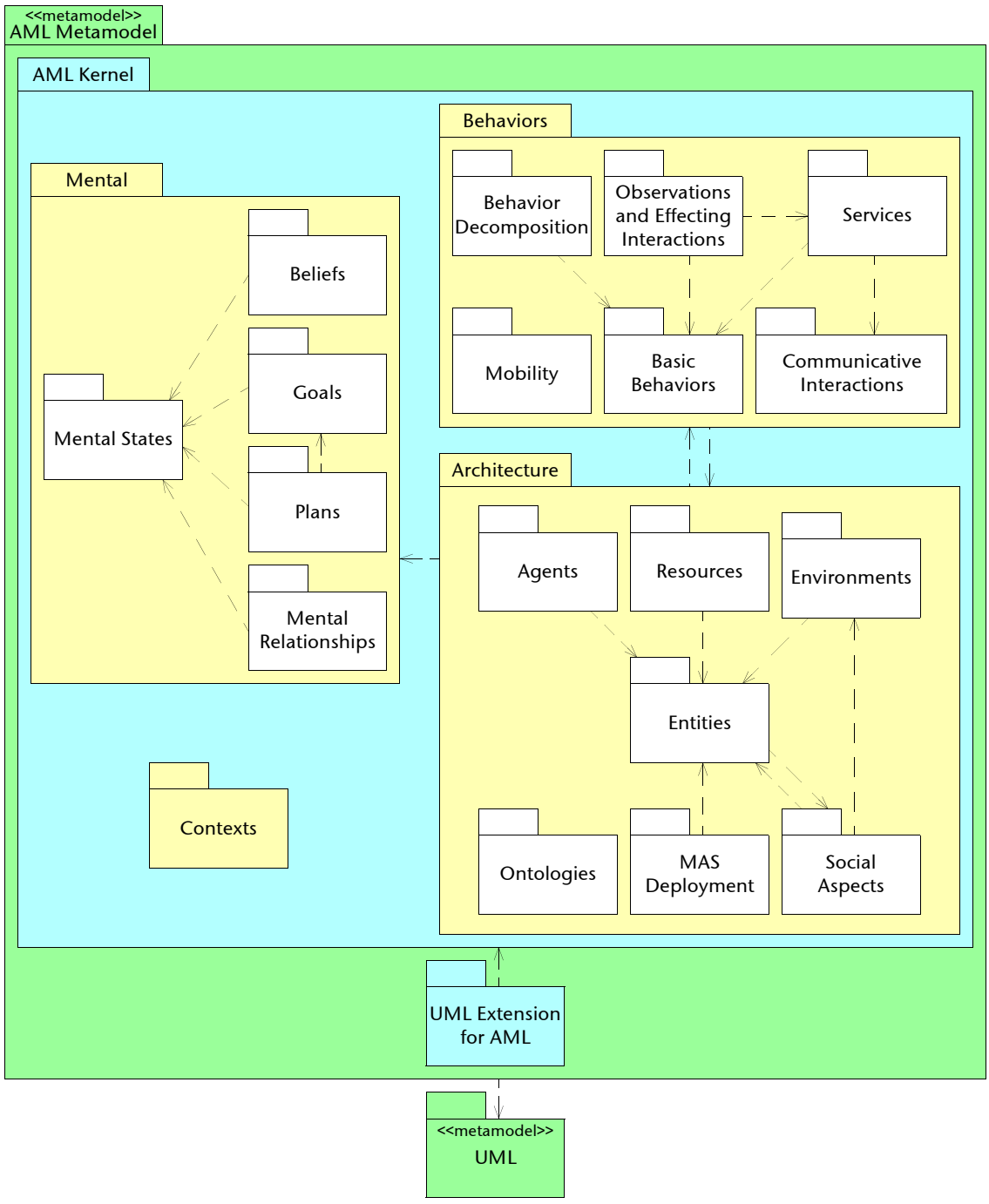


Fig. 3-1 Overall package structure of the AML metamodel

The AML Metamodel is logically structured according to the various aspects of MAS abstractions. All packages and their content are described in the following sections.

4 Architecture

Overview The *Architecture* package defines the metaclasses used to model architectural aspects of multi-agent systems.

Package structure The package diagram of the Architecture package is depicted in Fig. 4-1.

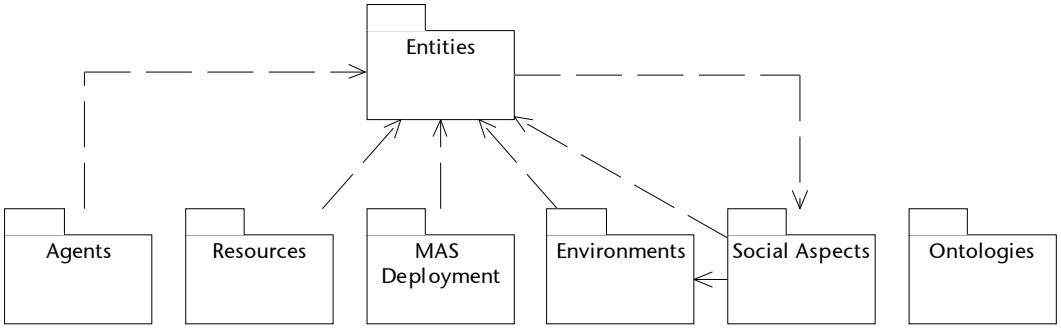


Fig. 4-1 Architecture—package structure

4.1 Entities

Overview The *Entities* package defines a hierarchy of abstract metaclasses that represent different kinds of AML entities. Entities are used to further categorize concrete AML metaclasses and to define their characteristic features.

Abstract syntax The diagram of the Entities package is shown in Fig. 4-2.

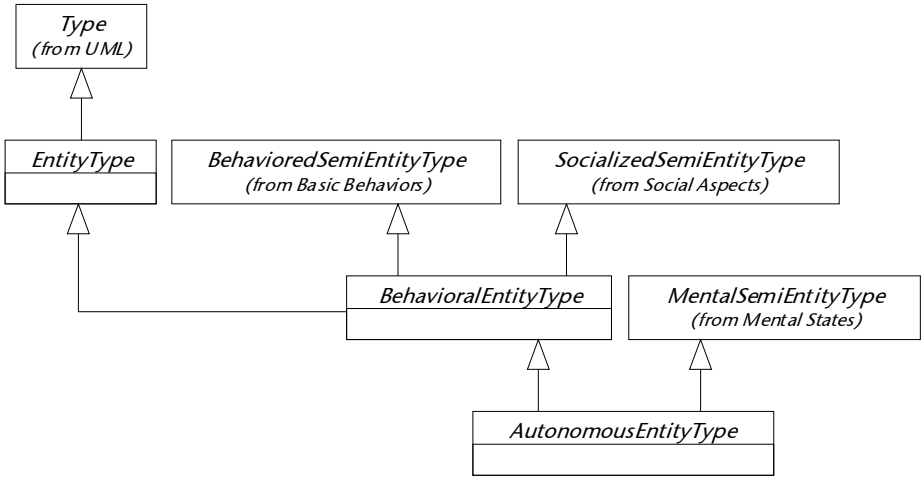


Fig. 4-2 Entities—hierarchy of entities

4.1.1 EntityType

Semantics EntityType is an abstract specialized Type (from UML). It is a superclass to all AML modeling elements which represent types of entities of a multi-agent system. *Entities* are understood to be objects, which can exist in the

system independently of other objects, e.g. agents, resources, environments.

EntityTypes can be hosted by AgentExecutionEnvironments (see sections *4.6.1 AgentExecutionEnvironment,* p. 49 and *4.6.2 HostingProperty,* p. 51), and can be mobile (see section *5.6.3 MobilityAction,* p. 131).

Notation There is no general notation for EntityType. The specific subclasses of EntityType define their own notation.

Rationale EntityType is introduced to allow explicit modeling of entities in the system, and to define the features common to all its subclasses.

4.1.2 BehavioralEntityType

Semantics BehavioralEntityType is an abstract specialized EntitiyType (p. 23) used to represent types of entities which have the features of BehavioredSemiEntityType (p. 60) and SocializedSemiEntityType (p. 33), and can play entity roles (see sections *4.5.7 RoleProperty,* p. 41 and *4.5.6 EntityRoleType,* p. 39).

Instances of BehavioralEntityTypes are referred to as *behavioral entities*.

Associations

/roleAttribute: RoleProperty[*]	A set of all RoleProperties owned by the BehavioralEntityType. It determines the EntityRoleTypes that may be played by the owning BehavioralEntityType. This association is ordered and derived. Subsets UML Class::ownedAttribute.

Constraints

1. The roleAttribute meta-association refers to all ownedAttributes of the kind RoleProperty:

   roleAttribute = self.ownedAttribute->select(oclIsKindOf(RoleProperty))

Notation BehavioralEntityType is generally depicted as UML Class, but the specific subclasses of BehavioralEntityType define their own notation.

Rationale BehavioralEntityType is introduced to define the features common to all its subclasses.

4.1.3 AutonomousEntityType

Semantics AutonomousEntityType is an abstract specialized BehavioralEntityType (p. 24) and MentalSemiEntityType (p. 143), used to model types of self-contained entities that are capable of autonomous behavior in their environment, i.e. entities that have control of their own behavior, and act upon their environment according to the processing of (reasoning on) perceptions of that environment, interactions and/or their mental attitudes. There are no other entities that directly control the behavior of autonomous entities.

AutonomousEntityType, being a MentalSemiEntityType, can be characterized in terms if its mental attitudes, i.e. it can own MentalProperties (p. 144).

Instances of AutonomousEntityTypes are referred to as *autonomous entities*.

Notation AutonomousEntityType is generally depicted as a UML Class, but the specific subclasses of AutonomousEntityType define their own notation.

Rationale AutonomousEntityType is introduced to allow explicit modeling of autonomous entities in the system, and to define the features common to all its subclasses.

4.2 Agents

Overview The *Agents* package defines the metaclasses used to model agents in multi-agent systems.

Abstract syntax The diagram of the Agents package is shown in Fig. 4-3.

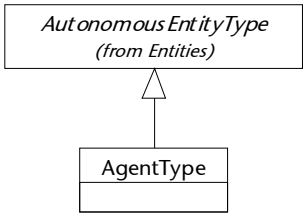


Fig. 4-3 Agents—agent type

4.2.1 AgentType

Semantics AgentType is a specialized AutonomousEntityType (p. 24) used to mode a type of *agents*, i.e. self-contained entities that are capable of autonomous behavior within their environment. An agent (instance of an AgentType) is a special object (which the OO paradigm defines as an entity having identity, status and behavior; not narrowed to a OO programming concept) having at least the following additional features[4]:

- ***Autonomy***, i.e. control over its own state and behavior, based on external (reactivity) or internal (proactivity) stimuli, and
- ***Ability to interact***, i.e. the capability to interact with its environment, including perceptions and effecting actions, speech act based interactions, etc.

AgentType can use all types of relationships allowed for UML Class, e.g. associations, generalizations, dependencies, etc., with their standard semantics (see [53]), as well as inherited AML-specific relationships described in further sections.

Note 1: An agent in AML represents an architectonic concept, that does not need to be necessarily implemented by a software or a physical agent. The

[4] Other features like *mobility*, *adaptability*, *learning*, etc., are optional in the AML framework.

implementation of AML agents can differ depending on the implementation environment (which does not necessary need to be an agent platform, or even a computer system).

Note 2: If required, potential AML extensions can define further subclasses of the AgentType metaclass in order to explicitly differentiate special types of agents. For instance: biological agent, human agent (specialized biological agent), artificial agent, software agent (specialized artificial agent), robot (specialized artificial agent), embedded system (specialized artificial agent), etc.

Notation AgentType is depicted as a UML Class with the stereotype <<agent>> and/or a special icon, see Fig. 4-4. All standard UML class compartments, user-defined compartments, internal structure of parts, ports, connectors, etc. (see UML StructuredClassifier), supported interfaces, provided and required services, owned behaviors, and a structure of the owned named elements can be specified as well. Their notation is described in further document sections or in UML 2.0 Superstructure [53].

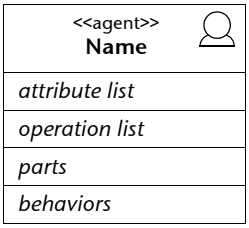


Fig. 4-4 Notation of AgentType

Examples Fig. 4-5 shows an example of the agent type named AgentSmith, having Capabilities (p. 62) findIntruder and convertToSmith, the internal structure composed of connected BehaviorFragments (p. 65), Perceptors (p. 119) eyes, and Effectors (p. 125) hands and legs.

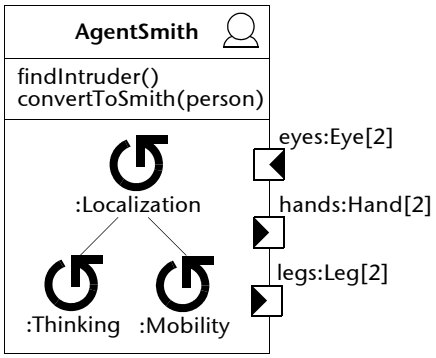


Fig. 4-5 Example of an AgentType

Rationale AgentType is introduced to model types of agents in multi-agent systems.

4.3 Resources

Overview The *Resources* package defines the metaclasses used to model resources in multi-agent systems.

Abstract syntax The diagram of the Resource package is shown in Fig. 4-6.

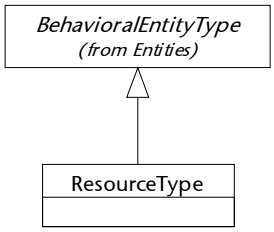


Fig. 4-6 Resources—resource type

4.3.1 ResourceType

Semantics ResourceType is a specialized BehavioralEntityType (p. 24) used to model types of resources contained within the system[5]. A *resource* is a physical or an informational entity, with which the main concern is its availability and usage (e.g. quantity, access rights, conditions of consumption).

Notation ResourceType is depicted as a UML Class with the stereotype <<resource>> and/or a special icon, see Fig. 4-7.

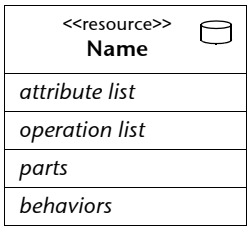


Fig. 4-7 Notation of ResourceType

Examples Fig. 4-8 shows an example of a usage of a ResourceType. An AgentType (p. 25) RPG_Player, representing a player of an RPG (Role Playing Game), owns a read-write resource inventory of the Inventory type. An Inventory has specified capacity and contains a set of items.

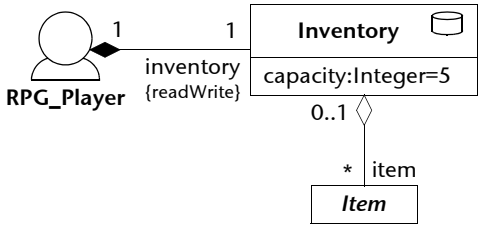


Fig. 4-8 Example of ResourceType

[5] A resource positioned outside a system is modeled as a UML Actor (or any subtype of an Actor).

Rationale ResourceType is introduced to model types of resources in multi-agent systems.

4.4 Environments

Overview The *Environments* package defines the metaclasses used to model system internal environments (for definition see section *4.4.1 EnvironmentType,* p. 28) of multi-agent systems.

Abstract syntax The diagram of the Environments package is shown in Fig. 4-9.

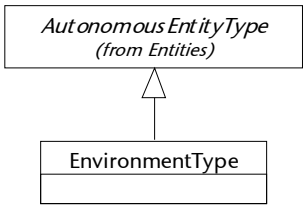


Fig. 4-9 Environments—environment type

4.4.1 EnvironmentType

Semantics EnvironmentType is a specialized AutonomousEntityType (p. 24) used to model types of *environments*, i.e. logical or physical surroundings of entities which provide conditions under which the entities exist and function. EnvironmentType thus can be used to define particular aspects of the world which entities inhabit, its structure and behavior. It can contain the space and all the other objects in the entity surroundings, but also those principles and processes (i.e. laws, rules, constraints, policies, services, roles, resources, etc.) which together constitute the circumstances under which entities act.

As environments are usually complex entities, different EnvironmentTypes are usually used to model different aspects of an environment.

From the point of view of the (multi-agent) system modeled, two categories of environments can be recognized:

- ***system internal environment***, which is a part of the system modeled, and
- ***system external environment***, which is outside the system modeled and forms the boundaries onto that system.

The EnvironmentType is used to model system internal environments, whereas system external environments are modeled by Actors (from UML).

An instance of the EnvironmentType is called *environment*.

One entity can appear in several environments at once and this set can dynamically change in time.

If required, the EnvironmentType can be extended by properties (tagged values) which can provide explicit classification of environments. For example, determinism, volatility, continuity, accessibility, etc. Definition of

such properties depends on specific needs and therefore is not part of the AML specification.

The EnvironmentType usually uses the possibilities inherited from StructuredClassifier, and models its internal structure by contained parts, connectors, ports, etc. All other relationship types defined for UML Class, and other inherited AML-specific relationships can be used for EnvironmentType as well.

Notation EnvironmentType is depicted as a UML Class with the stereotype <<environment>> and/or a special icon, see Fig. 4-10.

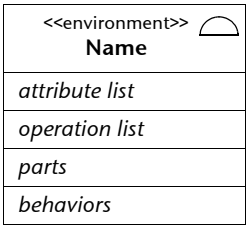


Fig. 4-10 Notation of EnvironmentType

Examples Fig. 4-11 shows a definition of an abstract Class 3DObject that represents spatial objects, characterized by shape and positions within containing spaces. An abstract EnvironmentType 3DSpace represents a three dimensional space. This is a special 3DObject and as such can contain other spatial objects.

Three concrete 3DObjects are defined: an AgentType (p. 25) Person, a ResourceType (p. 27) Ball and a Class named Goal. 3DSpace is furthermore specialized into a concrete EnvironmentType Pitch representing a soccer pitch containing two goals and a ball.

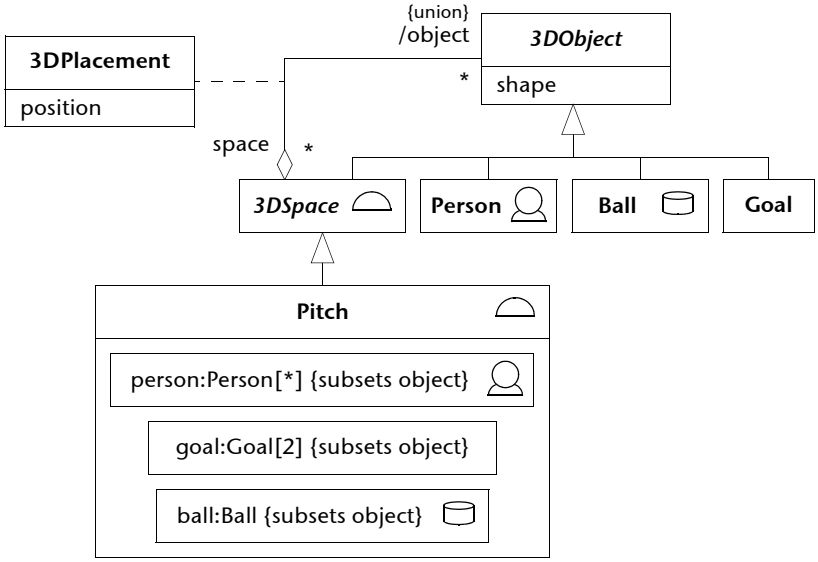


Fig. 4-11 Example of EnvironmentType

Rationale EnvironmentType is introduced to model particular aspects of the system internal environment.

4.5 Social Aspects

Overview The *Social Aspects* package defines metaclasses used to model abstractions of social aspects of multi-agent systems, including structural characteristics of socialized entities and certain aspects of their social behavior.

Abstract syntax The diagrams of the Social Aspects package are shown in figures Fig. 4-12 to Fig. 4-17.

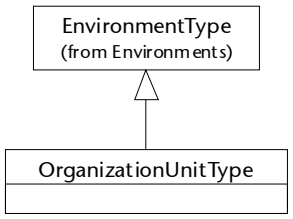


Fig. 4-12 Social Aspects—organization unit type

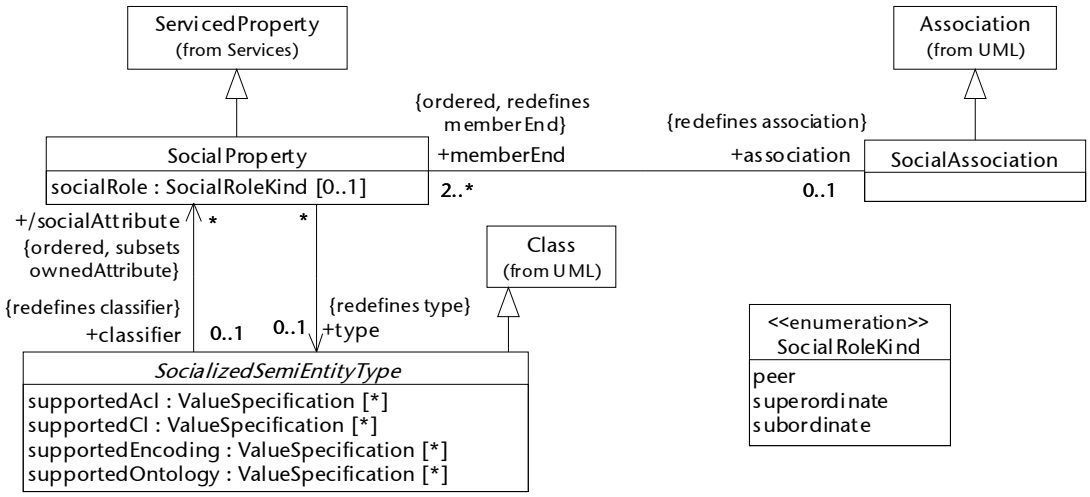


Fig. 4-13 Social Aspects—social property, social association, and socialized semi-entity type

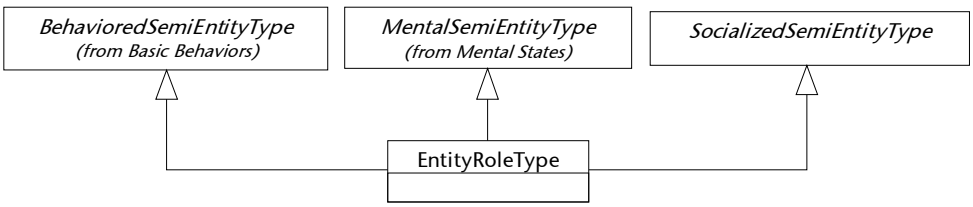


Fig. 4-14 Social Aspects—entity role type

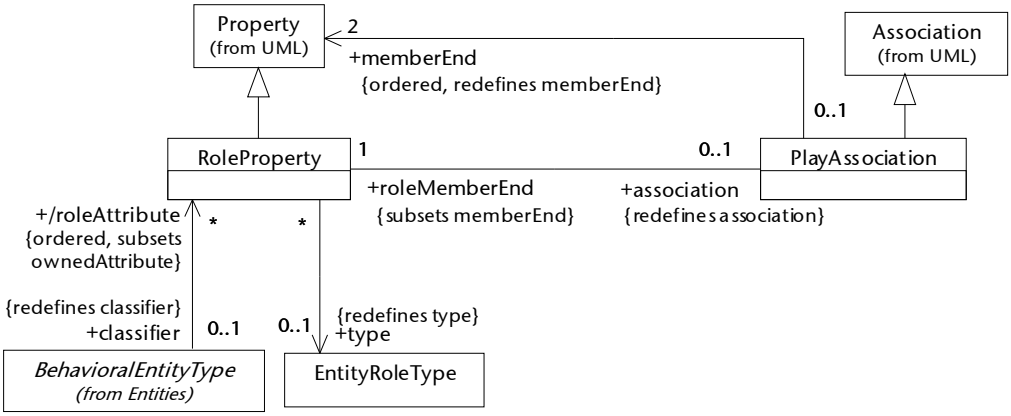


Fig. 4-15 Social Aspects—role property and play association

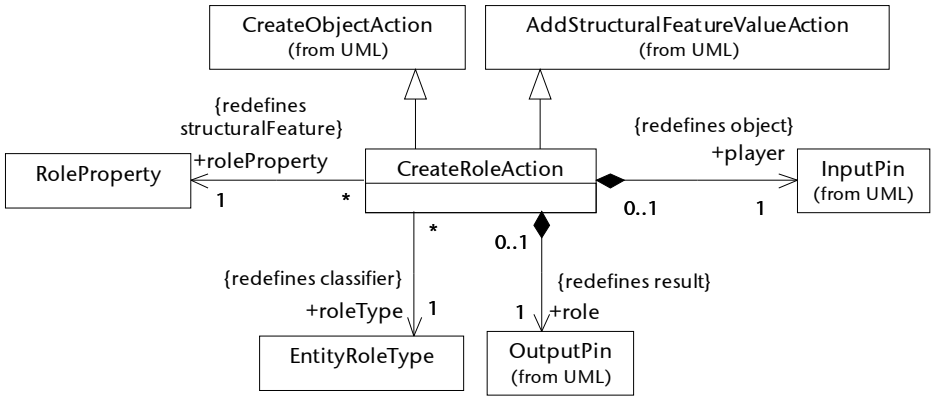


Fig. 4-16 Social Aspects—create role action

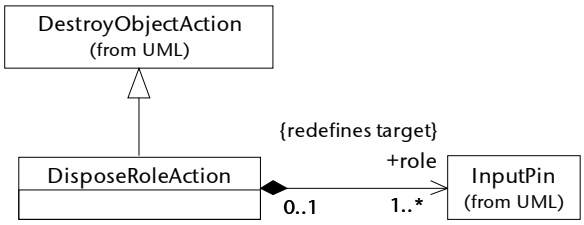


Fig. 4-17 Social Aspects—dispose role action

4.5.1 OrganizationUnitType

Semantics OrganizationUnitType is a specialized EnvironmentType (p. 28) used to model types of organization units, i.e. types of social environments or their parts.

An instance of the OrganizationUnitType is called *organization unit*.

From an ***external perspective***, organization units represent coherent autonomous entities which can have goals, perform behavior, interact with their environment, offer services, play roles, etc. Properties and behavior of organization units are both:

- emergent properties and behavior of all their constituents, their mutual relationships, observations and interactions, and
- the features and behavior of organization units themselves.

From an ***internal perspective***, organization units are types of environment that specify the social arrangements of entities in terms of structures, interactions, roles, constraints, norms, etc.

Notation OrganizationUnitType is depicted as a UML Class with the stereotype <<organization unit>> and/or a special icon, see Fig. 4-18.

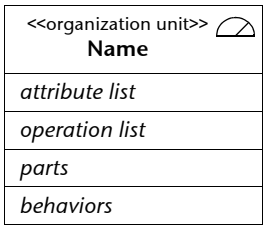


Fig. 4-18 Notation of OrganizationUnitType

Style guidelines An OrganizationUnitType is usually depicted in the form of a UML StructuredClassifier, and the contained entities (entity roles, agents, resources, environments, their relationships, etc.) as its parts, ports and connectors.

Examples Fig. 4-19 shows the definition of an OrganizationUnitType called SoccerTeam which represents a simplified model of a soccer team during a match. It groups several entity roles (for details see section *4.5.6 EntityRoleType,* p. 39) and their social relationships modeled by connectors of special types (for details see section *4.5.5 SocialAssociation,* p. 38).

A soccer team contains seven to eleven playing players and one to three coaches. The coaches lead the players and the players cooperate with each other.

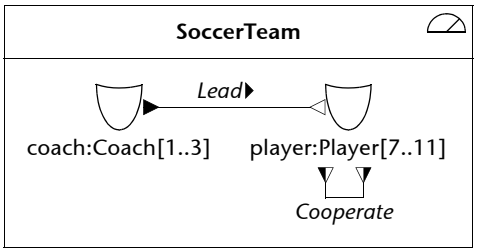


Fig. 4-19 Example of OrganizationUnitType's internal structure

OrganizationUnitType is used to model organization structures. Fig. 4-20 shows a class diagram depicting a generic organization structure of soft-

ware development project. Types of project roles, teams, and their social associations are shown.

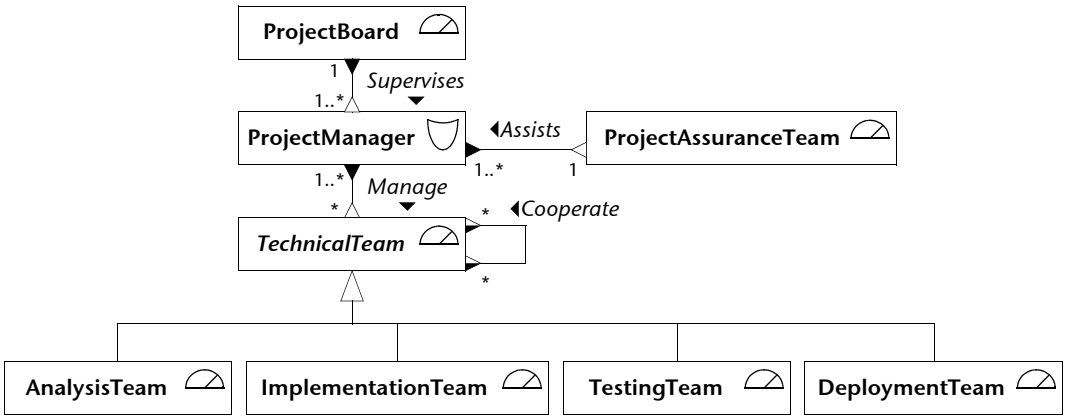


Fig. 4-20 Organization structure example—definition of types

The types defined in the previous organization diagram are used to model the internal structure of the SoftwareDevelopmentProject, see Fig. 4-21. Additionally, this OrganizationUnitType offers two external services (see section *5.4 Services,* p. 101).

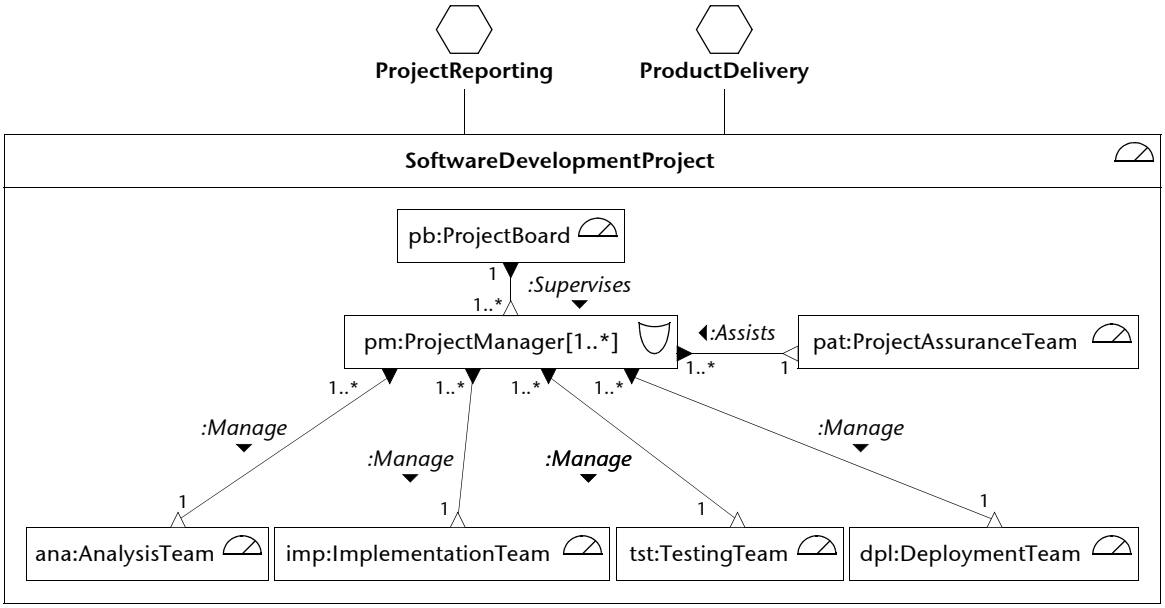


Fig. 4-21 Organization structure example—internal class structure

Rationale OrganizationUnitType is introduced to model types of organization units in multi-agent systems.

4.5.2 SocializedSemiEntityType

Semantics SocializedSemiEntityType is an abstract specialized Class (from UML), a superclass to all metaclasses which can participate in SocialAssociatons (p. 38) and can own SocialProperties (p. 35). There are two direct subclasses of the

SocializedSemiEntityType: BehavioralEntityType (p. 24) and EntityRoleType (p. 39).

SocializedSemiEntityTypes represent modeling elements, which would most likely participate in CommunicativeInteractions (p. 89). Therefore they can specify meta-attributes related to the CommunicativeInteractions, particularly: a set of agent communication languages (supportedAcl), a set of content languages (supportedCl), a set of message content encodings (supportedEncoding), and a set of ontologies (supportedOntology) they support. This set of meta-attributes can be extended by AML users if needed.

Instances of SocializedSemiEntityTypes are referred to as *socialized semi-entities*.

Attributes

supportedAcl: ValueSpecification[*]	A set of supported agent communication languages.
supportedCl: ValueSpecification[*]	A set of supported content languages.
supportedEncoding: ValueSpecification[*]	A set of supported message content encodings.
supportedOntology: ValueSpecification[*]	A set of supported ontologies.

Associations

/socialAttribute: SocialProperty[*]	A set of all SocialProperties owned by the SocializedSemiEntityType. This association is ordered and derived. Subsets UML Class::ownedAttribute.

Constraints

1. The socialAttribute meta-association refers to all ownedAttributes of the kind SocialProperty:

   ```
   socialAttribute = self.ownedAttribute->
      select(oclIsKindOf(SocialProperty))
   ```

Notation

There is no general notation for SocializedSemiEntityType. The specific subclasses of the SocializedSemiEntityType define their own notation.

The meta-attributes are shown as a property string of the owning SocializedSemiEntityType. The following keywords are used:

- acl,
- cl,
- encoding,
- ontology.

Their values represent arbitrary lists of ValueSpecifications, but the most commonly used types are enumerations or string literals.

Examples An example of a specification of the meta-attributes for an agent type called Surgeon is depicted in Fig. 4-22.

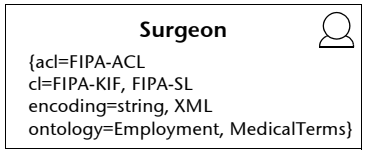


Fig. 4-22 Example of specification of meta-attributes for a SocializedSemiEntityType

Rationale SocializedSemiEntityType is introduced to define the features common to all its subclasses.

4.5.3 SocialProperty

Semantics SocialProperty is a specialized ServicedProperty (p. 109) used to specify social relationships that can or must occur between instances of its type and:

- ❒ instances of its owning class (when the SocialProperty is an attribute of a Class), or
- ❒ instances of the associated class (when the SocialProperty is a member end of an Association).

SocialProperty can be only of a SocializedSemiEntityType (p. 33) type.

SocialProperties can be owned only by:

- ❒ SocializedSemiEntityTypes as attributes, or
- ❒ SocialAssociations (p. 38) as member ends.

When a SocialProperty is owned by a SocializedSemiEntityType, it represents a *social attribute*. In this case the SocialProperty can explicitly declare a social role of its type in regard to the owning class. The social role of the owning SocializedSemiEntityType in regard to the social property's type is implicitly derived according to the rules described in Tab. 4-1.

Social role of attribute's type	***Derived social role of attribute owner***
peer	*peer*
superordinate	*subordinate*
subordinate	*superordinate*

Tab. 4-1 Rules for determining a social role of the SocialProperty owner

Ownership of a SocialProperty by a SocializedSemiEntityType can be considered as an implicit declaration of a binary SocialAssociation between owning SocializedSemiEntityType and a type of the attribute. It is usually used to model social relationships of structured socialized classes (e.g. EnvironmentTypes) and their parts.

When a SocialProperty is owned by a SocialAssociation, it represents a non-navigable end of the association. In this case the SocialProperty declares a social relation of its type (connected SocializedSemiEntityType) in regard to the other association end types (SocializedSemiEntityTypes connected to the other association ends).

Attributes

socialRole: SocialRoleKind[0..1]	The kind of a social relationship between SocialProperty's type and the owning SocializedSemiEntityType.

Associations

association: SocialAssociation[0..1]	The owning SocialAssociation of which this SocialProperty is a member, if any. Redefines UML Property::association.
type: SocializedSemiEntityType[0..1]	The type of a SocialProperty. Redefines UML TypedElement::type.

Constraints

1. If SocialProperty is a member end of a SocialAssociation and its socialRole is set to *peer*, the socialRoles of all other member ends must be set to *peer* as well:

   (self.association->notEmpty() and self.socialRole=#peer) implies
   self.association.memberEnd->forAll(socialRole=#peer)

2. If SocialProperty is a member end of a SocialAssociation and its socialRole is set to *superordinate*, the socialRoles of all other member ends must be set to *subordinate*:

   (self.association->notEmpty() and self.socialRole=#superordinate)
   implies self.association.memberEnd->
   forAll(me|me <> self and me.socialRole=#subordinate)

3. If SocialProperty is a member end of a SocialAssociation and its socialRole is set to *subordinate*, the socialRole of some another member end must be set to *superordinate*:

   (self.association->notEmpty() and self.socialRole=#subordinate) implies
   self.association.memberEnd->exists(socialRole=#superordinate)

Notation

When shown as an association end, the SocialProperty is depicted as a UML association end with a stereotype specifying the social role kind. The keyword is defined by the SocialRoleKind enumeration. See Fig. 4-23.

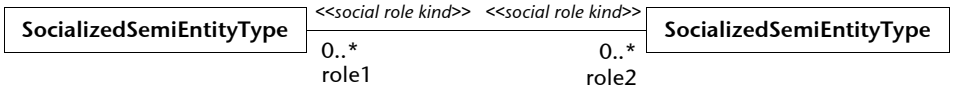


Fig. 4-23 Notation of SocialProperty shown as an association end

When shown as an attribute, the SocialProperty is depicted as a UML attribute with an additional keyword specifying the social role kind. The keyword is defined by the SocialRoleKind enumeration. See Fig. 4-24.

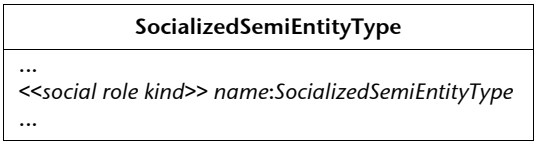


Fig. 4-24 Notation of SocialProperty shown as an attribute

Presentation options An iconic notation for particular social role kinds can be used instead of textual labels. Alternative notation is depicted in Fig. 4-25.

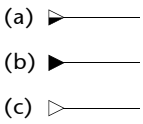


Fig. 4-25 Alternative notation for SocialProperty shown as an association end: *(a) peer, (b) superordinate, (c) subordinate.*

The same icons can be used also for a SocialProperty specified as an attribute. See Fig. 4-26.

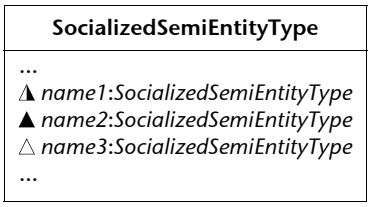


Fig. 4-26 Alternative notation of SocialProperty shown as an attribute

Examples Fig. 4-27 (a) shows an example of peer-to-peer social relationship between sellers and buyers. In case (b), an OrganizationUnitType (p. 31) Market comprises many market members who are subordinated to their home market. The case (c) is semantically similar to the case (b), but the Market is depicted as a StructuredClassifier and its market members are represented as a part.

Rationale SocialProperty is introduced to model social relationships between entities in multi-agent systems.

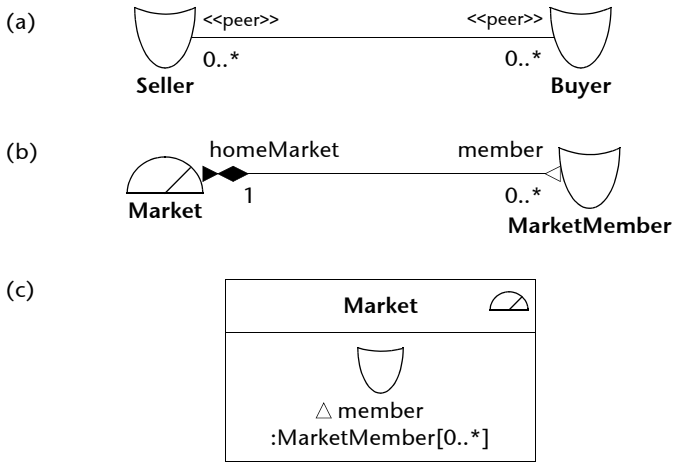


Fig. 4-27 Examples of SocialProperty

4.5.4 SocialRoleKind

Semantics SocialRoleKind is an enumeration which specifies allowed values for the socialRole meta-attribute of the SocialProperty (p. 35).

AML supports modeling of superordinate-subordinate and peer-to-peer relationships, but this set can be extended as required (e.g. to model producer-consumer, competition, or cooperation relationships).

Enumeration values Tab. 4-2 specifies SocialRoleKind's enumeration literals, stereotypes used for notation and their semantics.

Value	*Stereotype*	*Semantics*
peer	<<peer>>	A social role kind used in the peer-to-peer relationships of the entities with the same social status and equal authority.
superordinate	<<super>>	A social role kind specifying higher authority and power for its owner than for other associated subordinate entities. Superordinate entity is able to constrain the behavior of its subordinates.
subordinate	<<sub>>	A social role kind specifying lower authority and power than associated superordinate entity.

Tab. 4-2 SocialRoleKind***'s enumeration literals***

Rationale SocialRoleKind is introduced to define allowed values for the socialRole meta-attribute of the SocialProperty.

4.5.5 SocialAssociation

Semantics SocialAssociation is a specialized Association (from UML) used to model social relationships that can occur between SocializedSemiEntityTypes (p. 33). It redefines the type of the memberEnd property of Association to SocialProperty (p. 35).

An instance of the SocialAssociation is called *social link*.

Associations

memberEnd: SocialProperty[2..*]	At least two SocialProperties representing participation of socialized semi-entities in a social link. This is an ordered association. Redefines UML Association::memberEnd.

Notation SocialAssociation is depicted as a UML Association with stereotype <<social>>. Ends of SocialAssociation can use a special notation that is described in the SocialProperty section. Notation of binary SocialAssociation is shown in Fig. 4-28.

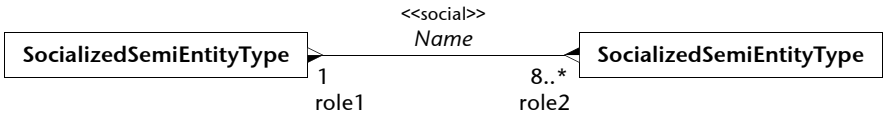


Fig. 4-28 Notation of binary SocialAssociation

Notation of n-ary social association is shown in Fig. 4-29.

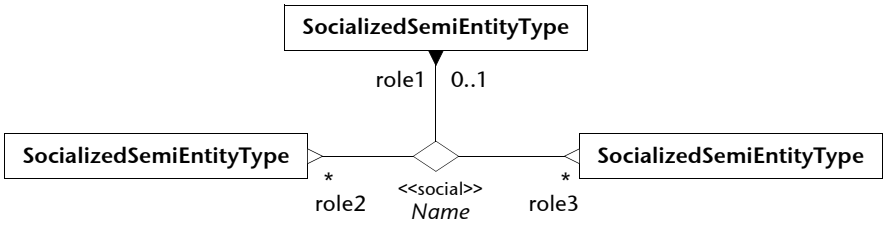


Fig. 4-29 Notation of n-ary SocialAssociation

Style guidelines The Association's stereotype is usually omitted if the association ends are stereotyped by social role kinds.

Examples Fig. 4-30 shows a peer-to-peer SocialAssociation named Deal of two entity role types Seller and Buyer.

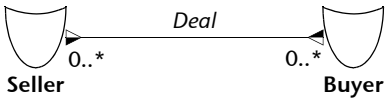


Fig. 4-30 Example of SocialAssociation

Other examples of SocialAssociations can be found in Fig. 4-11 (p. 29), Fig. 4-19 (p. 32), Fig. 4-20 (p. 33), Fig. 4-27 (p. 38), Fig. 4-33 (p. 41), and Fig. 4-40 (p. 45).

Rationale SocialAssociation is introduced to model social relationships between entities in multi-agent systems in the form of an Association.

4.5.6 EntityRoleType

Semantics EntityRoleType is a specialized BehavioredSemiEntityType (p. 60), MentalSemiEntityType (p. 143), and SocializedSemiEntityType (p. 33), used to rep-

resent a coherent set of features, behaviors, participation in interactions, and services offered or required by BehavioralEntityTypes (p. 24) in a particular context (e.g. interaction or social).

Each EntityRoleType thus should be defined within a specific larger behavior (collective behavior) which represents the context in which the EntityRoleType is defined together with all the other behavioral entities it interacts with. An advisable means to specify collective behaviors in AML is to use EnvironmentType (p. 28) or Context (p. 171).

Each EntityRoleType should be realized by a specific implementation possessed by a BehavioralEntityType which may play that EntityRoleType (for details see sections *4.5.7 RoleProperty,* p. 41 and *4.5.8 PlayAssociation,* p. 43).

EntityRoleType can be used as an indirect reference to behavioral entities, and as such can be utilized for the definition of reusable patterns.

An instance of an EntityRoleType is called *entity role*[6]. It represents either an execution of a behavior, or usage of features, or participation in interactions defined for the particular EntityRoleType by a behavioral entity (see section *4.1.2 BehavioralEntityType,* p. 24 for details). The entity role exists only while a behavioral entity plays it.

An entity role represented in a model as an InstanceSpecification classified by an EntityRoleType represents an instance of any BehavioralEntityType that may play that EntityRoleType, if the player (a behavioral entity which plays that entity role) is not specified explicitly.

When an EntityRoleType is used to specify the type of a TypedElement, values represented by that TypedElement are constrained to be instances of those BehavioralEntityTypes that may play given EntityRoleType.

An EntityRoleType, composed of other EntityRoleTypes (i.e. owning aggregated attributes having types of EntityRoleType), represents a *position type*. Its instantiation results in particular *position*s. A position can also be expressed implicitly by specifying several entity roles representing a specific position to be played by one behavioral entity holding this position.

Notation EntityRoleType is depicted as a UML Class with the stereotype <<entity role>> and/or a special icon, see Fig. 4-31.

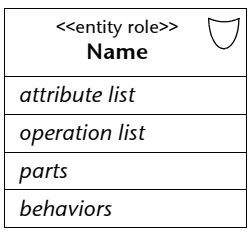


Fig. 4-31 Notation of EntityRoleType

[6] AML uses term ‘entity role’ to differentiate agent-related roles from the roles defined by UML 2.0 Superstructure, i.e. roles used for collaborations, parts, and associations.

Examples Fig. 4-32 shows an example of an EntityRoleType called SoccerPlayer. It defines structural properties (attributes, association roles, perceptors and effectors), behavioral features (operations), and usage of services. It does not specify methods (Behaviors) for the contained operations, therefore they must be defined by concrete BehavioralEntityTypes that play the SoccerPlayer EntityRoleType.

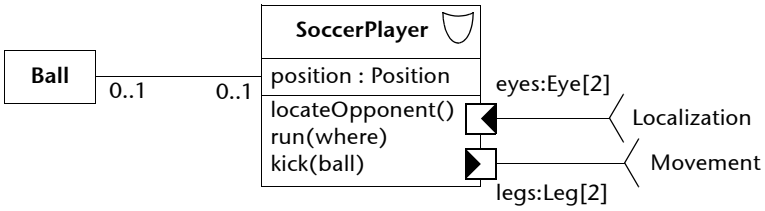


Fig. 4-32 EntityRoleType example—soccer player

Fig. 4-33 shows instantiation of EntityRoleTypes. In this example an agent Oliver plays an entity role master (depicted as an instantiated PlayAssociation). An agent John holds an implicitly specified position of a driver and a valet which can be possibly played simultaneously. Slots shown for the driver entity role specify its property values. An agent Mary plays an entity role cook. The figure shows also the links which represent instances of social associations defined on respective EntityRoleTypes. The corresponding class diagram is depicted in Fig. 4-37.

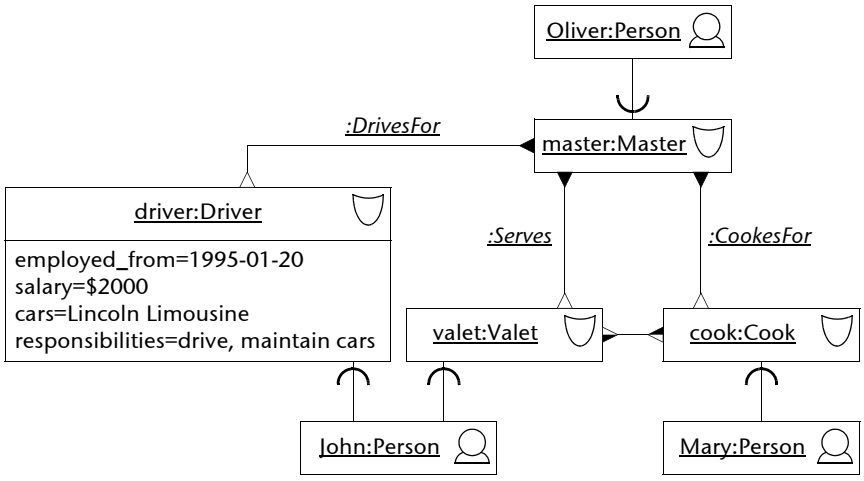


Fig. 4-33 EntityRoleType example—instances

Other examples of EntityRoleTypes can be found in Fig. 4-11 (p. 29), Fig. 4-19 (p. 32), and Fig. 4-20 (p. 33).

Rationale EntityRoleType is introduced to model roles in multi-agent systems.

4.5.7 RoleProperty

Semantics RoleProperty is a specialized Property (from UML) used to specify that an instance of its owner, a BehavioralEntityType (p. 24), can play one or several entity roles of the specified EntityRoleType (p. 39).

The owner of a RoleProperty is responsible for implementation of all Capabilities (p. 62), StructuralFeatures and metaproperties defined by SocializedSemiEntityType (p. 33) which are defined by RoleProperty's type (an EntityRoleType).

Instances of the played EntityRoleType represent (can be substituted by) instances of the RoleProperty owner.

One behavioral entity can at each time play (instantiate) several entity roles. These entity roles can be of the same as well as of different types. The multiplicity defined for the RoleProperty constraints the number of entity roles of given type, the particular behavioral entity can play concurrently.

Associations

association: PlayAssociation[0..1]	The owning PlayAssociation of which this RoleProperty is a member, if any. Redefines UML Property::association.
type: EntityRoleType[0..1]	The type of a RoleProperty. Redefines UML TypedElement::type.

Constraints

1. The aggregation meta-attribute of the RoleProperty is composite:

   self.aggregation = #composite

Notation

When shown as the end of a PlayAssociation (p. 43), the RoleProperty is depicted as a UML association end.

When shown as an attribute, the RoleProperty is depicted as a UML attribute with the stereotype <<role>>, see Fig. 4-34.

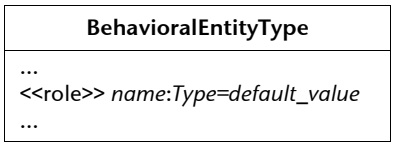


Fig. 4-34 Notation of RoleProperty shown as an attribute

Presentation options

The role properties of a BehavioralEntityType can be placed in a special class compartment named <<roles>>. The stereotype <<role>> of a particular RoleProperty is in this case omitted. See Fig. 4-35.

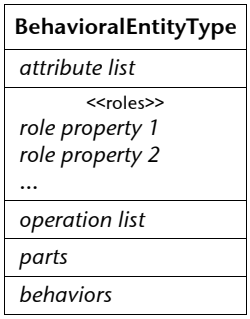


Fig. 4-35 Alternative notation for RoleProperty placed in a special class compartment

Examples Fig. 4-36 shows an example of specifying RoleProperties as attributes in a special class compartment. An agent Person can play four kinds of entity roles: master, valet, driver and cook.

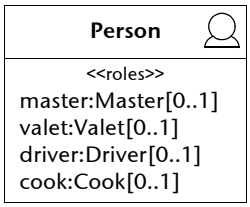


Fig. 4-36 Example of RoleProperties specified as attributes

Fig. 4-37 shows a semantically identical model, but the RoleProperties are specified as ends of PlayAssociations.

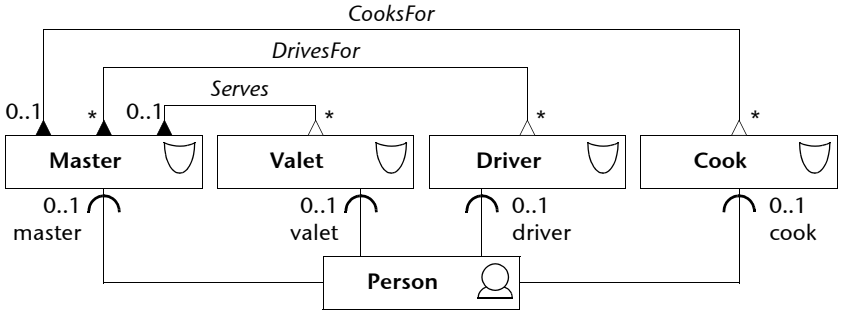


Fig. 4-37 Example of RoleProperties specified as association ends

Rationale RoleProperty is introduced to model the possibility of playing entity roles by behavioral entities.

4.5.8 PlayAssociation

Semantics PlayAssociation is a specialized Association (from UML) used to specify RoleProperty (p. 41) in the form of an association end. It specifies that entity roles of a roleMemberEnd's type (which is an EntityRoleType, p. 39) can be played, i.e. instantiated by entities of the other end type (which are BehavioralEntityTypes, p. 24).

Each entity role can be played by at most one behavioral entity. Therefore:

- The multiplicity of the PlayAssociation at the BehavioralEntityType side is always 0..1, and thus is not shown in diagrams.
- PlayAssociations attached to one EntityRoleType are implicitly constrained by an n-ary logical XOR operator. These constraints are implicit and thus not shown in diagrams.

Multiplicity on the entity role side of the PlayAssociation constrains the number of entity roles the particular BehavioralEntityType can instantiate concurrently.

An instance of the PlayAssociation is called *play link*.

Other notation parts defined for UML Association (e.g. qualifier, property string, navigability, etc.) can be specified for the PlayAssociation as well. Their semantics are specified in UML 2.0 Superstructure [53].

Associations

memberEnd: Property[2]	Two associated Properties. This is an ordered association. Redefines UML Association::memberEnd.
roleMemberEnd: RoleProperty[1]	Associated RoleProperty. Subsets PlayAssociation::memberEnd.

Notation PlayAssociation is depicted as a UML Association with the <<play>> stereotype, see Fig. 4-38. The multiplicity at the BehavioralEntityType side is unspecified.

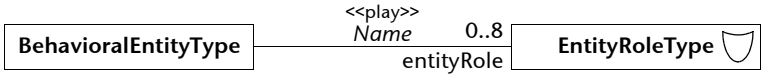


Fig. 4-38 Notation of PlayAssociation

Presentation options Instead of a stereotyped Association, the PlayAssociation can be depicted as an association line with a small thick semicircle as an arrow head at the end of EntityRoleType. See Fig. 4-39.

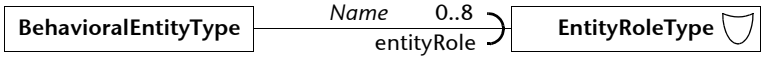


Fig. 4-39 Alternative notation of PlayAssociation

Style guidelines Stereotype labels or icons of association ends are usually omitted because their end types determine them implicitly. An association end connected to an EntityRoleType represents a RoleProperty and therefore the <<role>> stereotype may be omitted.

Even if the aggregation meta-attribute of the RoleProperty is always composite, a solid filled diamond is usually not shown in the PlayAssociation.

Examples Fig. 4-40 shows an example where each instance of an AgentType (p. 25) Person and an OrganizationUnitType (p. 31) Bank can play several entity roles of type Buyer. Each of these entity roles is used to specify a deal with a specific seller.

Rationale PlayAssociation is introduced to model the possibility of playing entity roles by behavioral entities.

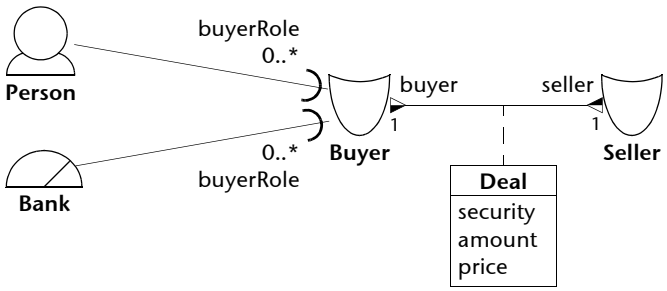


Fig. 4-40 Example of PlayAssociation relationship

4.5.9 CreateRoleAction

Semantics

CreateRoleAction is a specialized CreateObjectAction (from UML) and AddStructuralFeatureValueAction (from UML), used to model the action of creating and starting to play an entity role by a behavioral entity.

Technically this is realized by instantiation of an EntityRoleType (p. 39) into an entity role of that type, and adding this instance as a value to the RoleProperty (p. 41) of its player (a behavioral entity) which starts to play it.

The CreateRoleAction specifies:

- what EntityRoleType is being instantiated (roleType meta-association),
- the entity role being created (role meta-association),
- the player of created entity role (player meta-association), and
- the RoleProperty owned by the type of player, where the created entity role is being placed (roleProperty meta-association).

If the RoleProperty referred to by the roleProperty meta-association is ordered, the insertAt meta-association (inherited from the AddStructuralFeatureValueAction) specifies a position at which to insert the entity role.

Because the value meta-association (inherited from UML WriteStructuralFeatureAction) represents the same entity role as is already represented by the role meta-association, the properties of the InputPin referred to by the value meta-association are ignored in CreateRoleAction, and can be omitted in its specification.

Associations

roleType: EntityRoleType[1]	Instantiated EntityRoleType. Redefines UML CreateObjectAction::classifier.
role: OutputPin[1]	The OutputPin on which the created entity role is put. Redefines UML CreateObjectAction::result.
player: InputPin[1]	The InputPin specifying the player of the created entity role. Redefines UML StructuralFeatureAction::object.

roleProperty: RoleProperty[1]	The RoleProperty where the created entity role is being placed. Redefines UML StructuralFeatureAction::structuralFeature.

Constraints

1. If the type of the InputPin referred to by the player meta-association is specified, it must be a BehavioralEntityType:

   self.player.type->notEmpty() implies
   self.player.type.oclIsKindOf(BehavioralEntityType)

2. If the type of the OutputPin referred to by the role meta-association is specified, it must conform to the EntityRoleType referred to by the roleType meta-association:

   self.role.type->notEmpty() implies
   self.role.type.conformsTo(self.roleType)

3. If the type of the RoleProperty referred to by the roleProperty meta-association is specified, the EntityRoleType referred to by the roleType meta-association must conform to it:

   self.roleProperty.type->notEmpty() implies
   self.roleType.conformsTo(self.roleProperty.type)

Notation

CreateRoleAction is shown as a UML Action with the stereotype <<create role>> and/or a special icon, see Fig. 4-41.

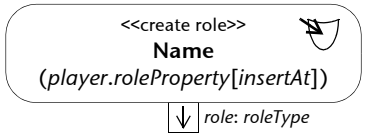


Fig. 4-41 Notation of CreateRoleAction

Optionally, the name of the player, delimited by a period from the name of the RoleProperty referred to by the roleProperty meta-association, may be specified in parentheses below the action's name. If the RoleProperty is ordered, the value of the insertAt meta-association can be placed after the RoleProperty's name in brackets.

If the player itself executes the CreateRoleAction, it can be identified by the keyword 'self' in place of player name.

A created entity role is specified as an OutputPin. All notational variations for the UML OutputPin are allowed. The EntityRoleType referred to by the roleType meta-association is specified as the type of the OutputPin.

A mandatory InputPin referred to by the value meta-association has unspecified properties and is not drawn in diagrams.

Examples

Fig. 4-42 shows an example of the activity describing the scenario of creating and matching a sell order at the stock exchange.

The activity comprises three activity partitions: an agent marketMember representing a Person who is playing an entity role of type MarketMember, organization unit stockExchange, and an entity role buyer.

The marketMember, wanting to sell some securities, creates a sellOrder and submits it to the stockExchange. The stockExchange registers this order and creates a new seller entity role (we can say: “a new seller position is opened at the stock exchange”). The marketMember becomes a seller by this registration. Then the stockExchange tries to match the sellOrder. If the matching was unsuccessful, the scenario terminates. If the matching was successful, the marketMember (now also seller) and the buyer concurrently confirm the match. When complete, the stockExchange performs a settlement and both the seller and buyer entity roles are disposed.

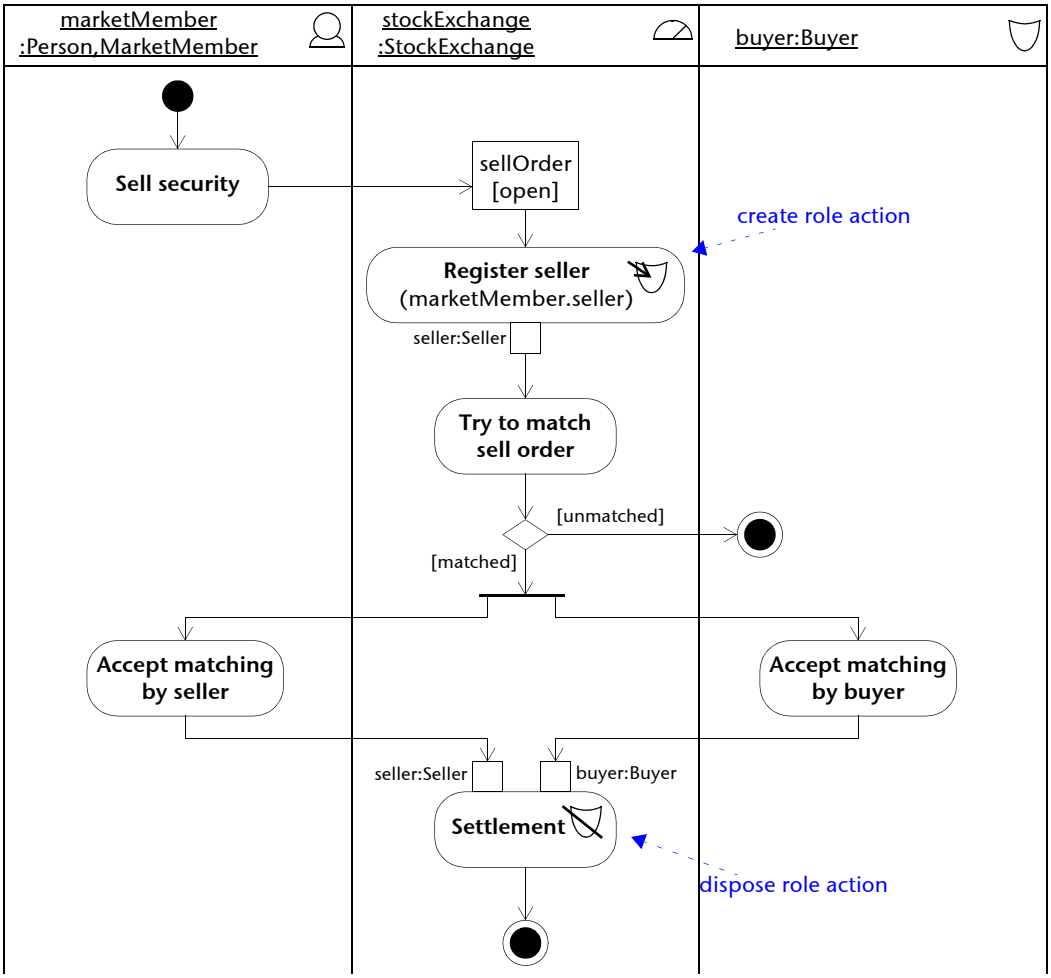


Fig. 4-42 Example of CreateRoleAction and DisposeRoleAction

Rationale CreateRoleAction is introduced to model an action of creating and playing entity roles by behavioral entities.

4.5.10 DisposeRoleAction

Semantics DisposeRoleAction is a specialized DestroyObjectAction (from UML) used to model the action of stopping to play an entity role by a behavioral entity.

Technically it is realized by destruction of the corresponding entity role(s). As a consequence, all behavioral entities that were playing the destroyed entity roles stop to play them.

Associations

role: InputPin[1..*]	The InputPins representing the entity roles to be disposed. Redefines UML DestroyObjectAction::target.

Constraints

1. If the types of the InputPins referred to by the role meta-association are specified, they must be EntityRoleTypes:

   self.role->forAll(ro | ro.type->notEmpty() implies
   ro.type.oclIsKindOf(EntityRoleType))

Notation DisposeRoleAction is drawn as a UML Action with the stereotype <<dispose role>> and/or a special icon, see Fig. 4-43. Disposed entity roles are depicted as InputPins.

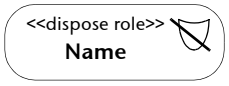


Fig. 4-43 Notation of DisposeRoleAction

Examples See Fig. 4-42 (p. 47).

Rationale DisposeRoleAction is introduced to model the action of disposing of entity roles by behavioral entities.

4.6 MAS Deployment

Overview The *MAS Deployment* package defines the metaclasses used to model deployment of a multi-agent system to a physical environment.

Abstract syntax The diagrams of the MAS Deployment package are shown in figures Fig. 4-44 and Fig. 4-45.

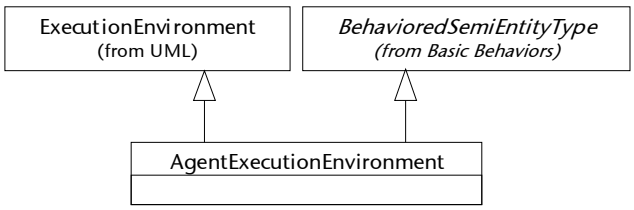


Fig. 4-44 MAS Deployment—agent execution environment

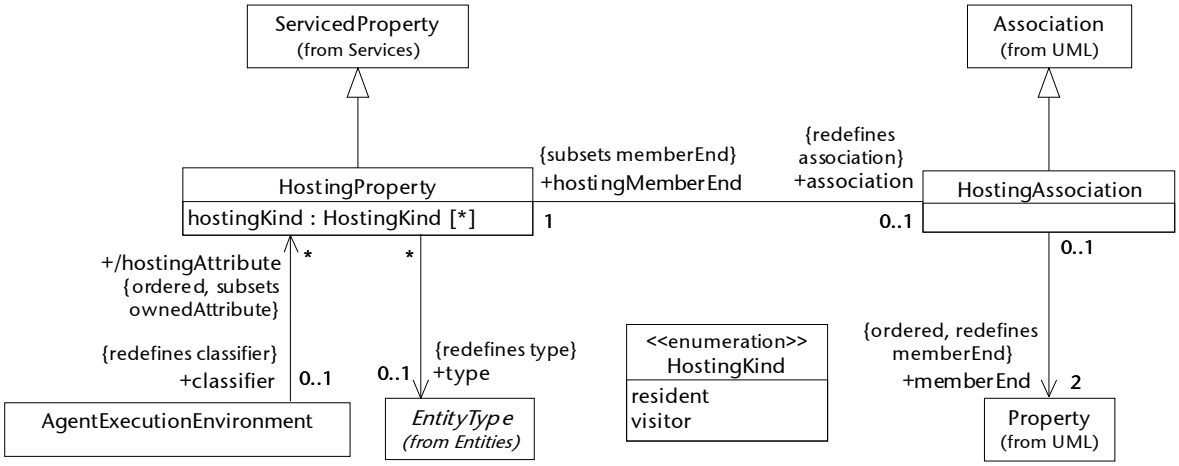


Fig. 4-45 MAS Deployment—hosting

4.6.1 AgentExecutionEnvironment

Semantics

AgentExecutionEnvironment is a specialized ExecutionEnvironment (from UML) and BehavioredSemiEntityType (p. 60), used to model types of execution environments of multi-agent systems. AgentExecutionEnvironment thus provides the physical infrastructure in which MAS entities can run. One entity can run at most in one AgentExecutionEnvironment instance at one time.

If useful, it may be further subclassed into more specific agent execution environments, e.g. agent platform, agent container, etc.

AgentExecutionEnvironment can provide (use) a set of services that deployed entities use (provide) at run time. AgentExecutionEnvironment, being a BehavioredSemiEntityType, can explicitly specify such services by means of ServiceProvisions (p. 112) and ServiceUsages (p. 114) respectively.

Owned HostingProperties (p. 51) specify kinds of entities hosted by (running at) the AgentExecutionEnvironment. Internal structure of the AgentExecutionEnvironment can also contain other features and behaviors that characterize it.

Associations

/hostingAttribute: HostingProperty[*]	A set of all HostingProperties owned by the AgentExecutionEnvironment. This association is ordered and derived. Subsets UML Class::ownedAttribute.

Constraints

1. The internal structure of an AgentExecutionEnvironment can also consist of other attributes than parts of the type Node. The constraint [1] defined for UML Node, and inherited by the AgentExecutionEnvironment, is therefore discarded (see [53] for details).
2. The hostingAttribute meta-association refers to all ownedAttributes of the kind HostingProperty:

```
hostingAttribute = self.ownedAttribute->
  select(oclIsKindOf(HostingProperty))
```

Notation AgentExecutionEnvironment is depicted as a UML ExecutionEnvironment with the stereotype <<agent execution environment>> and/or a special icon, see Fig. 4-46.

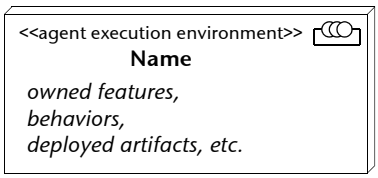


Fig. 4-46 Notation of AgentExecutionEnvironment

Examples Fig. 4-47 shows the example of an MAS deployment model. A Node called StockExchangeServer runs an AgentExecutionEnvironment of type TradingServer and this in turn enables the hosting of agents of type Broker (modeled by a HostingProperty), resources of type Account and one environment of type OrderPool. A Node ClientPC runs an AgentExecutionEnvironment of type TradingClient in which agents of type Broker can be hosted. All hosting entities are modeled as hostingAttributes.

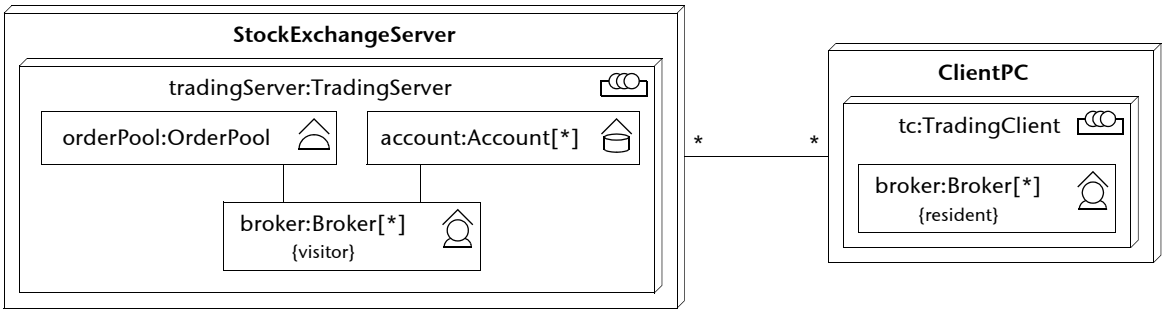


Fig. 4-47 MAS deployment example—class diagram

Fig. 4-48 shows an instance-level example of the AgentExecutionEnvironment called AgentPlatform, distributed over several Nodes. Each Node contains its own specific AgentContainer, where the local agents operate. The mainNode, in addition to the main agent container mainAc, has also deployed the central database centralDB used by the agent container.

For another example see also Fig. 5-117 (p. 130).

Rationale AgentExecutionEnvironment is introduced to model execution environments of multi-agent systems, i.e. the environments in which the entities exist and operate.

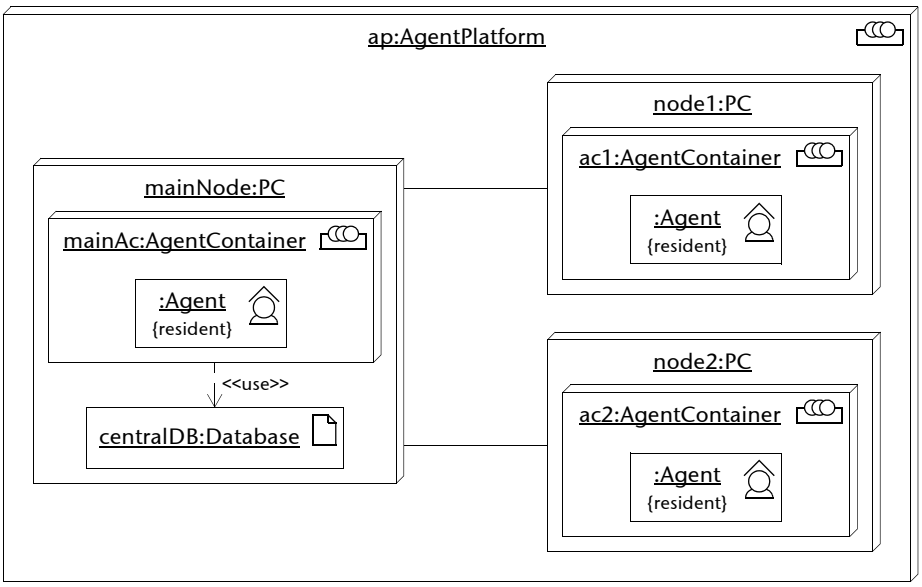


Fig. 4-48 MAS deployment example—instance diagram

4.6.2 HostingProperty

Semantics

HostingProperty is a specialized ServicedProperty (p. 109) used to specify what EntityTypes (p. 23) can be hosted by what AgentExecutionEnvironments (p. 49).

Type of a HostingProperty can be only an EntityType.

HostingProperties can be owned only by:

- ❒ AgentExecutionEnvironments as attributes, or
- ❒ HostingAssociations (p. 54) as member ends.

The owned meta-attribute hostingKind specifies the relation of the referred EntityType to the owning AgentExecutionEnvironment (for details see section *4.6.3 HostingKind,* p. 54).

Attributes

hostingKind: HostingKind[*]	A set of hosting kinds that the owning AgentExecutionEnvironment provides to the HostingProperty's type.

Associations

association: HostingAssociation [0..1]	The owning HostingAssociation of which this HostingProperty is a member, if any. It represents a hosting place. Redefines UML Property::association.

type: EntityType[0..1]	The type of a HostingProperty. It represents an entity that resides at the hosting AgentExecutionEnvironment. Redefines UML TypedElement::type.
/move: Move[*]	A set of the Move (p. 129) dependencies that refer to the HostingProperty as source of moving. This is a derived association.
/moveFrom: Move[*]	A set of the Move dependencies that refer to the HostingProperty as destination of moving. This is a derived association.
/clone: Clone[*]	A set of the Clone (p. 130) dependencies that refer to the HostingProperty as source of cloning. This is a derived association.
/cloneFrom: Clone[*]	A set of the Clone dependencies that refer to the HostingProperty as destination of cloning. This is a derived association.

Constraints

1. The move meta-association refers to all clientDependencies of the kind Move:

   move = self.clientDependency->select(oclIsKindOf(Move))

2. The moveFrom meta-association refers to all supplierDependencies of the kind Move:

   moveFrom = self.supplierDependency->select(oclIsKindOf(Move))

3. The clone meta-association refers to all clientDependencies of the kind Clone:

   clone = self.clientDependency->select(oclIsKindOf(Clone))

4. The cloneFrom meta-association refers to all supplierDependencies of the kind Clone:

   cloneFrom = self.supplierDependency->select(oclIsKindOf(Clone))

Notation

When shown as the end of a HostingAssociation, the HostingProperty is depicted as a UML association end (see section *4.6.4 HostingAssociation,* p. 54 for details).

When shown as an attribute, the HostingProperty is depicted as a UML attribute with the stereotype <<hosting>>, see Fig. 4-49.

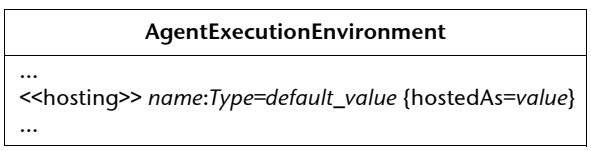


Fig. 4-49 Notation of HostingProperty shown as an attribute

The hostingKind meta-attribute is specified as a tagged value with the keyword 'hostedAs' and a value containing a list of HostingKind literals separated by ampersands ('&').

Alternatively, the HostingProperty can be depicted as a ConnectableElement with the stereotype <<hosting>>, see Fig. 4-50.

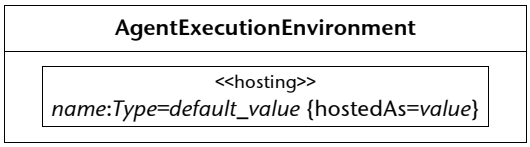


Fig. 4-50 Notation of HostingProperty shown as a ConnectableElement

Presentation options All HostingProperties of an AgentExecutionEnvironment can be placed in a special compartment named <<hosting>>. The stereotype <<hosting>> of a particular HostingProperty is in this case omitted. See Fig. 4-51.

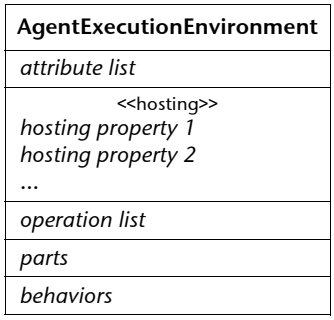


Fig. 4-51 Alternative notation for HostingProperty placed in a special class compartment

To recognize the fundamental type of HostingProperty's type visually and to specify just one stereotype, AML offers a notation shortcut to combine both stereotypes when the HostingProperty is depicted as a ConnectableElement. The basic shape of the modified stereotype icon represents an original EntityType's stereotype icon placed under a "shelter", which means hosting. Variants of stereotype icons for all concrete EntityTypes are depicted in Fig. 4-52.

Fig. 4-52 Alternative notation for HostingProperty depicting a stereotype of its type's fundamental type: *(a) hosting of an agent, (b) hosting of a resource, (c) hosting of an environment, (d) hosting of an organization unit.*

If the only displayed tagged value is the hostingKind, its keyword may be omitted and only its value is specified. Alternatively, the property string may be placed under the name/type string.

Examples See Fig. 4-47 (p. 50), Fig. 4-48 (p. 51), and Fig. 5-117 (p. 130).

Rationale HostingProperty is introduced to model the hosting of EntityTypes by AgentExecutionEnvironments.

4.6.3 HostingKind

Semantics HostingKind is an enumeration which specifies possible hosting relationships of EntityTypes (p. 23) to AgentExecutionEnvironments (p. 49). These are:

- ***resident***—the EntityType is perpetually hosted by the AgentExecutionEnvironment.
- ***visitor***—the EntityType can be temporarily hosted by the AgentExecutionEnvironment, i.e. it can be temporarily moved or cloned to the corresponding AgentExecutionEnvironment.

If needed, the set of available hosting kinds can be extended.

Enumeration values Tab. 4-3 specifies HostingKind's enumeration literals, keywords used for notation and their semantics.

Value	*Keyword*	*Semantics*
resident	resident	Instances of AgentExecutionEnvironment represent home for resident entities, i.e. execution environments where the entities are deployed and operate perpetually.
visitor	visitor	Instances of AgentExecutionEnvironment represent hosts for visiting entities, i.e. temporary operational environments.

Tab. 4-3 HostingKind***'s enumeration literals***

Rationale HostingKind is introduced to define possible values of the hostingKind meta-attribute of the HostingProperty (p. 51) metaclass.

4.6.4 HostingAssociation

Semantics HostingAssociation is a specialized Association (from UML) used to specify HostingProperty (p. 51) in the form of an association end. It specifies that entities classified according to a hostingMemberEnd's type (which is an EntityType, p. 23) can be hosted by instances of an AgentExecutionEnvironment (p. 49) representing the other end type.

HostingAssociation is a binary association.

An instance of the HostingAssociation is called *hosting link*.

Other notation parts defined for UML Association (e.g. qualifier, property string, navigability, etc.) can be specified for the HostingAssociation as well. Their semantics are specified in UML 2.0 Superstructure [53].

Associations

memberEnd: Property[2]	Two associated Properties. This is an ordered association. Redefines UML Association::memberEnd.

hostingMemberEnd: HostingProperty[1]	Associated HostingProperty. Subsets HostingAssociation::memberEnd.

Notation HostingAssociation is depicted as a UML Association with the <<hosting>> stereotype, see Fig. 4-53.

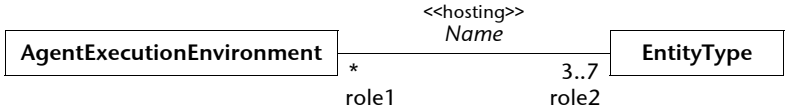


Fig. 4-53 Notation of HostingAssociation

Presentation options Instead of a stereotyped Association, the HostingAssociation can be depicted as an association line with a small thick "V" shape at the end of EntityRoleType, pointing in the direction of the AgentExecutionEnvironment. See Fig. 4-39.

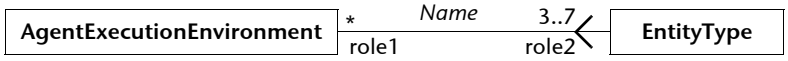


Fig. 4-54 Alternative notation of HostingAssociation

Examples Fig. 4-55 shows an example of hosting Broker, Account and an OrderPool EntityTypes by the TradingServer AgentExecutionEnvironment. Broker is hosted as a visitor.

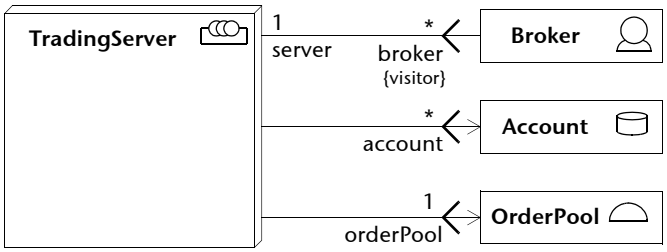


Fig. 4-55 Example of HostingAssociation

Rationale HostingAssociation is introduced to model the hosting of EntityTypes by AgentExecutionEnvironments in the form of an Association.

4.7 Ontologies

Overview The *Ontologies* package defines the metaclasses used to model ontologies. AML allows the specification of class-level as well as instance-level ontologies.

The basic means for modeling of ontologies in AML are ontology classes and their instances, their relationships, constraints, and ontology utilities. Ontology models are structured by means of the ontology packages.

Abstract syntax The diagram of the Ontology package is shown in Fig. 4-56.

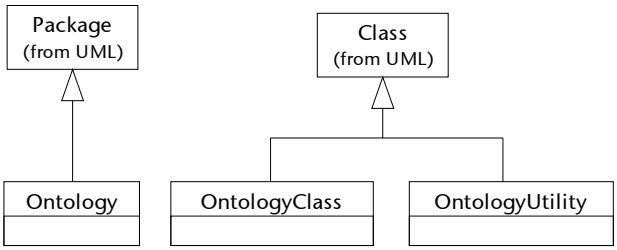


Fig. 4-56 Ontology—all elements

4.7.1 Ontology

Semantics Ontology is a specialized Package (from UML) used to specify a single ontology. By utilizing the features inherited from UML Package (package nesting, element import, package merge, etc.), Ontologies can be logically structured.

Notation Ontology is depicted as a UML Package with the stereotype <<ontology>> and/or a special icon, see Fig. 4-57.

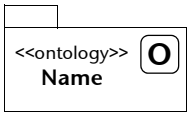


Fig. 4-57 Notation of Ontology

Style guidelines The ontology stereotype is usually displayed as decoration (a small icon placed in upper-right corner) of the original UML Package symbol.

Examples Fig. 4-58 shows an example of importing and merging of ontologies in the form of a package diagram.

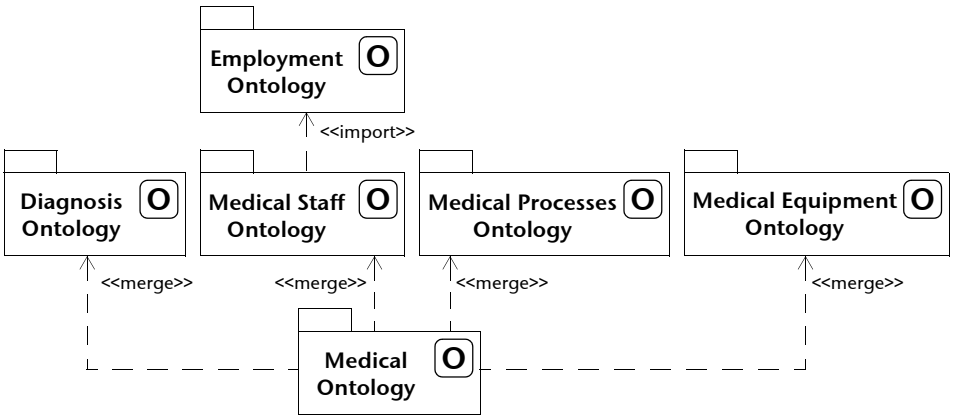


Fig. 4-58 Example of Ontology package diagram

Rationale Ontology is introduced to specify a single ontology.

4.7.2 OntologyClass

Semantics OntologyClass is a specialized Class (from UML) used to represent an ontology class (called also ontology concept or frame).

Attributes and operations of the OntologyClass represent its slots. Ontology functions, actions, and predicates belonging to a concept modeled by an OntologyClass are modeled by its operations.

OntologyClass can use all types of relationships allowed for UML Class (Association, Generalization, Dependency, etc.) with their standard UML semantics.

Even if UML predefines the "facet" used for attributes and operations (i.e. the form of metainformation that can be specified for them, e.g. name, multiplicity, list of parameters, return value, standard tagged values, etc.), the user is allowed to extend this metainformation by adding specific tagged values.

OntologyClass can also be used as an AssociationClass.

Notation OntologyClass is depicted as a UML Class with the stereotype <<oclass>> and/or a special icon, see Fig. 4-59.

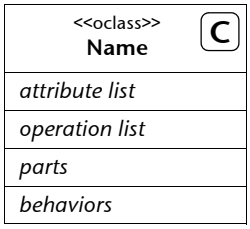


Fig. 4-59 Notation of OntologyClass

Examples Fig. 4-60 shows a fragment of the Medical Staff Ontology built up from several OntologyClasses.

Rationale OntologyClass is introduced to model an ontology class (also called concept or frame).

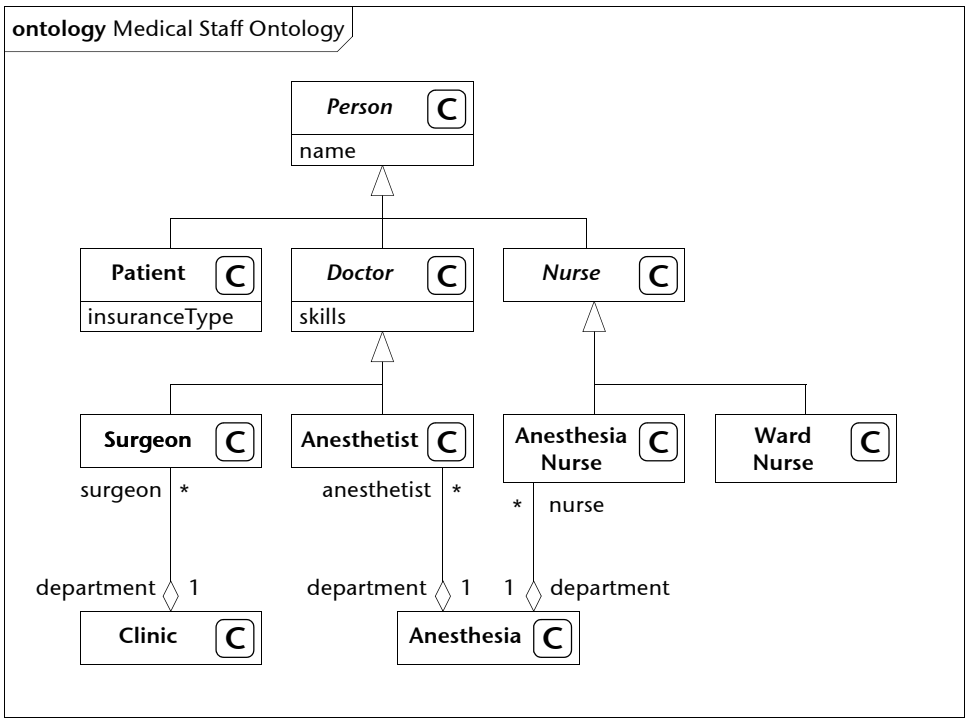


Fig. 4-60 Example of OntologyClass

4.7.3 OntologyUtility

Semantics OntologyUtility is a specialized Class (from UML) used to cluster global *ontology constants*, *ontology variables*, and *ontology functions/actions/predicates* modeled as owned features. The features of an OntologyUtility can be used by (referred to by) other elements within the owning and importing Ontologies (p. 56).

There can be more than one OntologyUtility classes within one Ontology. In such a way different OntologyUtilities provide clusters for logical grouping of their features.

OntologyUtility has no instances, all its features are class-scoped.

Notation OntologyUtility is depicted as a UML Class with the stereotype <<outility>> and/or a special icon, see Fig. 4-61.

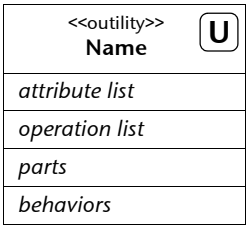


Fig. 4-61 Notation of OntologyUtility

Examples Fig. 4-62 shows a Goniometry ontology utility. It contains the PI constant and three ontology functions sin(x), cos(x), and tan(x). The Goniometry ontology utility is defined within the Math Ontology.

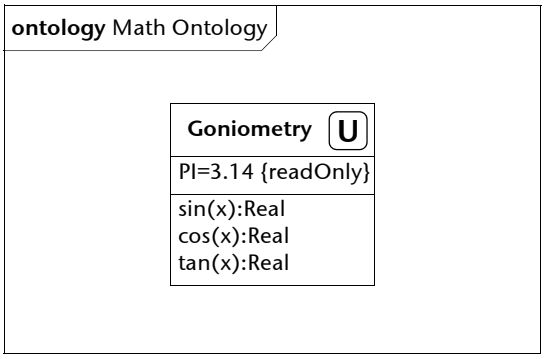


Fig. 4-62 Example of OntologyUtility

Rationale OntologyUtility is introduced to cluster global ontology constants, ontology variables, and ontology functions/actions/predicates.

5 Behaviors

Overview The *Behaviors* package defines the AML metaclasses used to model behavioral aspects of multi-agent systems.

Package structure The package diagram of the Behaviors package is depicted in Fig. 5-1.

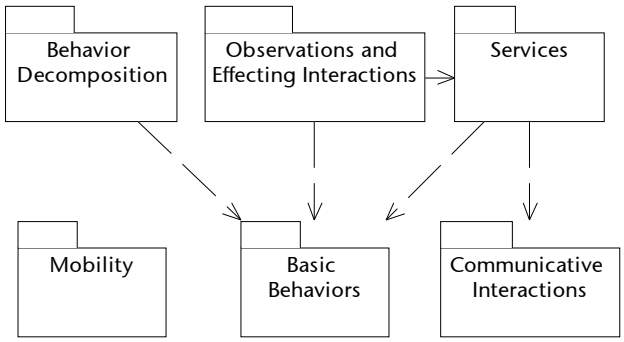


Fig. 5-1 Behaviors—package structure

5.1 Basic Behaviors

Overview The *Basic Behaviors* package defines the core, frequently referred metaclasses used to model behavior in AML.

Abstract syntax The diagram of the Basic Behaviors package is shown in Fig. 5-2.

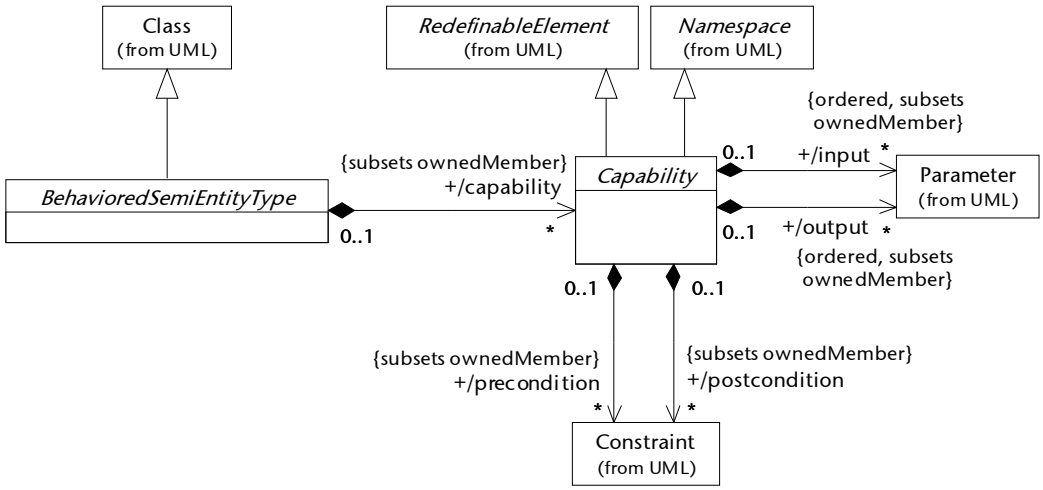


Fig. 5-2 Basic Behaviors—behaviored semi-entity type and capability

5.1.1 BehavioredSemiEntityType

Semantics BehavioredSemiEntityType is an abstract specialized Class (from UML) and ServicedElement (p. 108), that serves as a common superclass to all metaclasses which can:

- own Capabilities (p. 62),
- observe and/or effect their environment by means of Perceptors (p. 119) and Effectors (p. 125), and
- provide and/or use services by means of ServicedPorts (p. 110).

Furthermore, behavior of BehavioredSemiEntityTypes (and related features) can be explicitly (and potentially recursively) decomposed into BehavioralFragments (p. 65).

In addition to the services provided and used directly by the BehavioredSemiEntityType (see the serviceUsage and the serviceProvision meta-associations inherited from the ServicedElement), it is also responsible for implementation of the services specified by all ServiceProvisions (p. 112) and ServiceUsages (p. 114) owned by the ServicedProperties (p. 109) and ServicedPorts having the BehavioredSemiEntityType as their type.

Instances of BehavioredSemiEntityTypes are referred to as *behaviored semi-entities*.

Associations

/capability: Capability[*]	A set of all Capabilities owned by a BehavioredSemiEntityType. This is a derived association.
/behaviorFragment: BehaviorFragment[*]	A set of all BehaviorFragments that decompose a BehavioredSemiEntityType. This is a derived association.
/ownedServicedPort: ServicedPort[*]	A set of all ServicedPorts owned by a BehavioredSemiEntityType. This is a derived association.
/ownedPerceptor: Perceptor[*]	All owned Perceptors. This is a derived association. Subsets BehavioredSemiEntityType::ownedServicedPort.
/ownedEffector: Effector[*]	All owned Effectors. This is a derived association. Subsets BehavioredSemiEntityType::ownedServicedPort.

Constraints

1. The capability meta-association is union of owned BehavioralFeatures and Behaviors:

   ```
   capability = self.ownedBehavior->union(self.feature->
     select(oclIsKindOf(BehavioralFeature)))
   ```

2. The behaviorFragment meta-association comprises types of all owned aggregate or composite attributes having the type of a BehaviorFragment:

   ```
   behaviorFragment = self.ownedAttribute->
     select(oa|(oa.aggregation=#shared or oa.aggregation=#composite)
     and oa.type->notEmpty()
     and oa.type.oclIsKindOf(BehaviorFragment)).type
   ```

3. The ownedServicedPort meta-association refers to all owned ports of the kind ServicedPort:

ownedServicedPort = self.ownedPort->select(oclIsKindOf(ServicedPort))

4. The ownedPerceptor meta-association refers to all owned service ports of the kind Perceptor:

   ownedPerceptor = self.ownedServicedPort->
   select(oclIsKindOf(Perceptor))

5. The ownedEffector meta-association refers to all owned service ports of the kind Effector:

   ownedEffector = self.ownedServicedPort->
   select(oclIsKindOf(Effector))

Notation There is no general notation for BehavioredSemiEntityType. The specific subclasses of BehavioredSemiEntityType define their own notation.

Rationale BehavioredSemiEntityType is introduced as a common superclass to all metaclasses which can have capabilities, can observe and/or effect their environment, and can provide and/or use services.

5.1.2 Capability

Semantics Capability is an abstract specialized RedefinableElement (from UML) and Namespace (from UML), used to model an abstraction of a behavior in terms of its inputs, outputs, pre-conditions, and post-conditions. Such a common abstraction allows use of the common features of all the concrete subclasses of the Capability metaclass uniformly, and thus reason about and operate on them in a uniform way.

To maintain consistency with UML, which considers pre-conditions as aggregates (see Operation and Behavior in *UML 2.0 Superstructure* [53]), all pre-conditions specified for one Capability are understood to be logically AND-ed to form a single logical expression representing an overall pre-condition for that Capability. This is analogously the case for post-conditions.

Capability, being a RedefinableElement, allows the redefinition of specifications (see UML Constraint::specification) of its pre- and post-conditions, e.g. when inherited from a more abstract Capability. Specification of redefined conditions are logically combined with the specification of redefining conditions (of the same kind), following the rules:

- ❒ overall pre-conditions are logically OR-ed, and
- ❒ overall post-conditions are logically AND-ed.

Input and output parameters must be the same for redefining Capability as defined in the context of redefined Capability.

The set of meta-attributes defined by the Capability can be further extended in order to accommodate specific requirements of users and/or implementation environments.

Capabilities can be owned by BehavioredSemiEntityTypes (p. 60).

Capability is part of the non-conservative extension of UML, while it is a common superclass to two UML metaclasses: BehavioralFeature and Behavior.

Associations

/input: Parameter[*]	An ordered list of input parameters of the Capability. This is a derived association. Subsets UML Namespace::ownedMember.
/output: Parameter[*]	An ordered list of output parameters of the Capability. This is a derived association. Subsets UML Namespace::ownedMember.
/precondition: Constraint[*]	An optional set of Constraints on the state of the system in which the Capability can be invoked. This is a derived association. Subsets UML Namespace::ownedMember.
/postcondition: Constraint[*]	An optional set of Constraints specifying the expected state of the system after the Capability is completed. This is a derived association. Subsets UML Namespace::ownedMember.

Constraints

1. The input meta-association refers to all parameters having the direction set either to *in* or *inout*:

```
input = if self.oclIsKindOf(BehavioralFeature) then
    self.oclAsType(BehavioralFeature).parameter->
    select(direction=#in or direction=#inout)
else
    self.oclAsType(Behavior).parameter->
    select(direction=#in or direction=#inout)
endif
```

2. The output meta-association refers to all parameters having the direction set either to *out* or *inout*:

```
output = if self.oclIsKindOf(BehavioralFeature) then
    self.oclAsType(BehavioralFeature).parameter->
    select(direction=#out or direction=#inout)
else
    self.oclAsType(Behavior).parameter->
    select(direction=#in or direction=#inout)
endif
```

3. The precondition meta-association is identical either to the precondition meta-association from Operation or the precondition meta-association from Behavior:

```
precondition = if self.oclIsKindOf(Behavior) then
    self.oclAsType(Behavior).precondition
else
    if self.oclIsKindOf(Operation) then
        self.oclAsType(Operation).precondition
```

```
    else
        Set{} -- self.oclIsKindOf(Reception)
    endif
endif
```

4. The postcondition meta-association is identical either to the postcondition meta-association from Operation or the postcondition meta-association from Behavior:

```
postcondition = if self.oclIsKindOf(Behavior) then
    self.oclAsType(Behavior).postcondition
else
    if self.oclIsKindOf(Operation) then
        self.oclAsType(Operation).postcondition
    else
        Set{} -- self.oclIsKindOf(Reception)
    endif
endif
```

Notation There is no general notation for Capability. The specific subclasses of Capability define their own notation.

Usually, the capability conditions are specified as a hidden information, not shown in the diagram. However, if a user needs to express them explicitly in the diagram, the conditions can be shown as a note symbol structured into sections specifying pre-conditions and post-conditions, see Fig. 5-3. Any of the sections can be omitted.

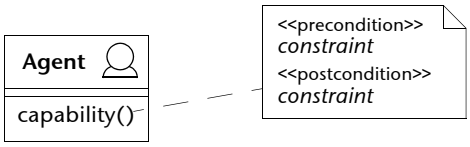


Fig. 5-3 Notation of the Capability meta-attribute specification

Presentation options Each single capability condition can be depicted as a separate note symbol connected to the Capability itself, see Fig. 5-4.

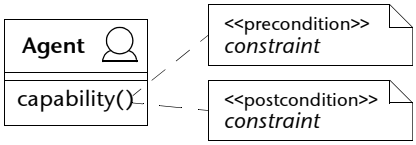


Fig. 5-4 Alternative notation of the Capability meta-attribute specification

Rationale Capability is introduced to define common meta-attributes for all “behavior-specifying” modeling elements in order to refer them uniformly, e.g. while reasoning.

5.2 Behavior Decomposition

Overview The *Behavior Decomposition* package defines the BehaviorFragment (p. 65) which allows the decomposition of complex behaviors of BehavioredSemiEntityTypes (p. 60) and the means to build reusable libraries of behaviors and related features.

Abstract syntax The diagram of the Behavior Decomposition package is shown in Fig. 5-5.

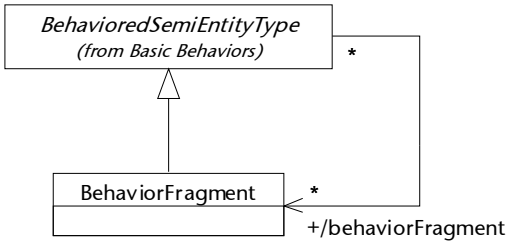


Fig. 5-5 Behavior Decomposition—behavior fragment

5.2.1 BehaviorFragment

Semantics BehaviorFragment is a specialized BehavioredSemiEntityType (p. 60) used to model coherent and reusable fragments of behavior and related structural and behavioral features, and to decompose complex behaviors into simpler and (possibly) concurrently executable fragments.

BehaviorFragments can be shared by several BehavioredSemiEntityTypes and a behavior of a BehavioredSemiEntityType can, possibly recursively, be decomposed into several BehaviorFragments.

The decomposition of a behavior of a BehavioredSemiEntityType to its sub-behaviors is modeled by owned aggregate attributes (having the aggregation meta-attribute set either to *shared* or *composite*) of the BehaviorFragment type. At runtime, the behaviored semi-entity delegates execution of its behavior to the containing BehaviorFragment instances.

Notation BehaviorFragment is depicted as a UML Class with the stereotype <<behavior fragment>> and/or a special icon, see Fig. 5-6.

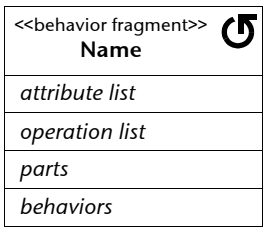


Fig. 5-6 Notation of BehaviorFragment

Style guidelines Decomposition of a BehavioredSemiEntityType into sub-behaviors is usually depicted as an aggregation or composition relating the BehavioredSemiEntityType to the BehaviorFragments, or as a structuring of the BehavioredSemiEntityType into parts (or attributes not owned by composition) of the BehaviorFragment type (see UML StructuredClassifier).

Examples Fig. 5-7 shows an example of a decomposition of an AgentType (p. 25) into BehaviorFragments. The behavior of the SoccerRobot AgentType is decomposed into four BehaviorFragments: Localization, Mobility, BallManipulation and SoccerPlayerThinking. Localization manages the robot's position in a pitch, enables it to observe surrounding objects, and to measure distances between them. It requires Observing and Measurement services provided by a pitch. Mobility allows movement within a physical space. BallManipulation is used to manipulate the ball. SoccerPlayerThinking comprises different strategies and ad-hoc behaviors of a soccer player. Strategy allows the execution of a certain strategy which influences global long-term behavior of a player. AdHocReaction allows short-term behaviors triggered by a particular situation, e.g. certain pre-learned tricks.

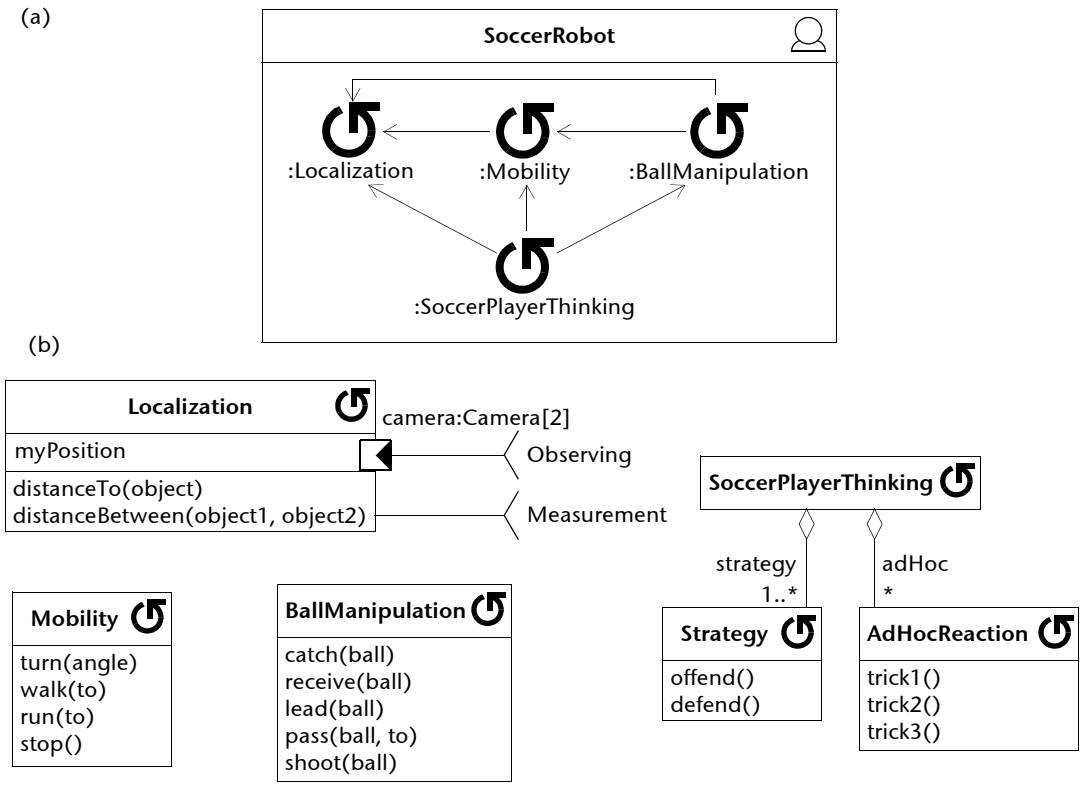


Fig. 5-7 Example of BehaviorFragment: *(a) decomposition of a behavior of an AgentType into BehaviorFragments, (b) definition of used BehaviorFragments.*

Another example of BehaviorFragment is in Fig. 4-5 (p. 26).

Rationale BehaviorFragment is introduced to: (a) decompose complex behaviors of BehavioredSemiEntities, and (b) build reusable libraries of behaviors and related features.

5.3 Communicative Interactions

Overview The *Communicative Interactions* package contains metaclasses that provide generic as well as agent specific extensions to UML Interactions.

The generic extension supports modeling of:

- ❒ interactions between groups of objects,
- ❒ dynamic change of an object's attributes induced by interactions, and
- ❒ messages not explicitly associated with an invocation of corresponding operations and signals.

The agent specific extension supports modeling of speech act based interactions between MAS entities and interaction protocols.

The focus of this section is mainly on Sequence Diagrams, however, notational variants for the Communication Diagrams are also mentioned.

Abstract syntax The diagrams of the Communicative Interactions package are shown in figures Fig. 5-9 to Fig. 5-18.

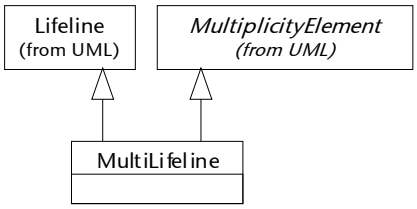


Fig. 5-8 Communicative Interactions—multi-lifeline

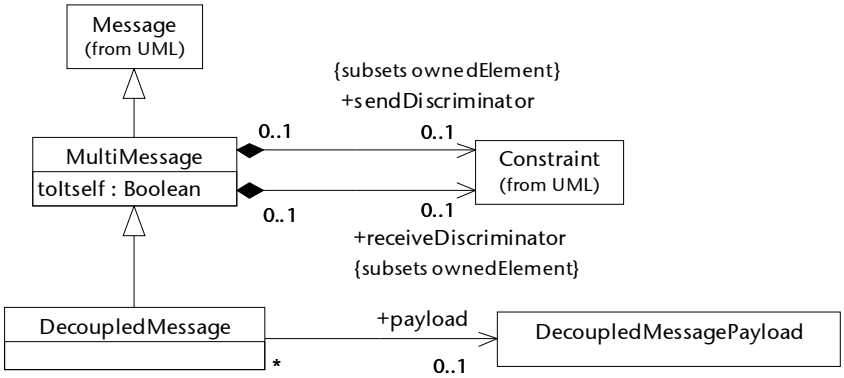


Fig. 5-9 Communicative Interactions—multi-message and decoupled message

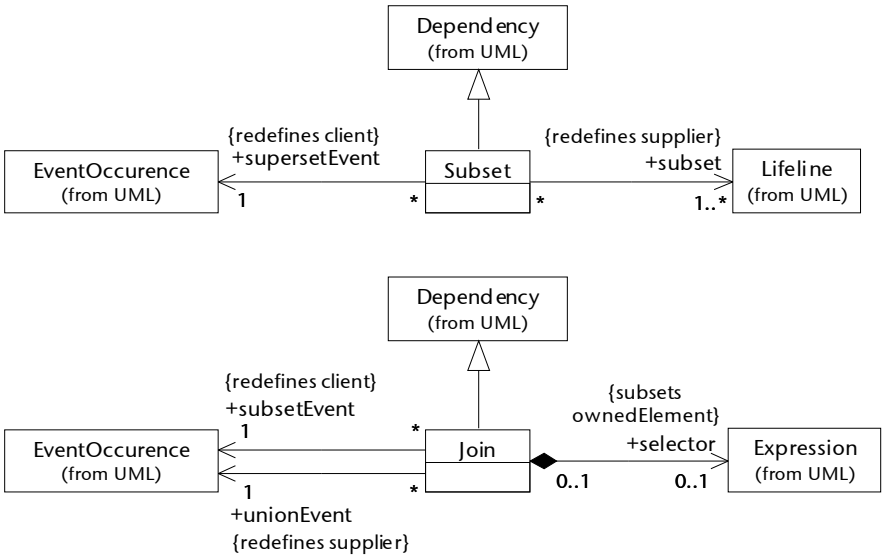


Fig. 5-10 Communicative Interactions—subset and join relationships

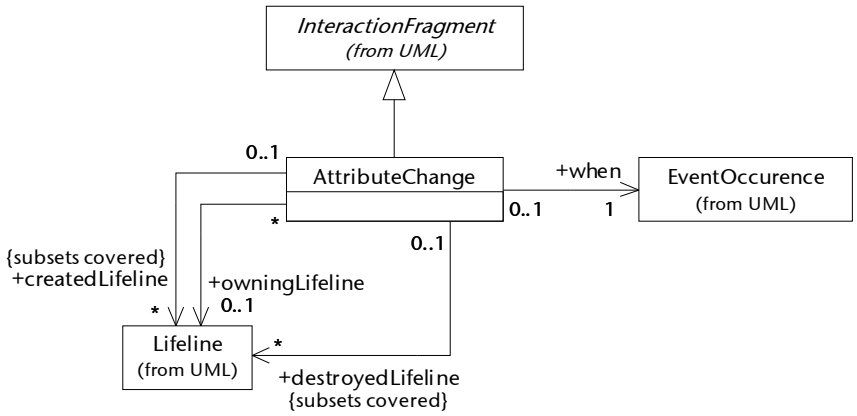


Fig. 5-11 Communicative Interactions—attribute change

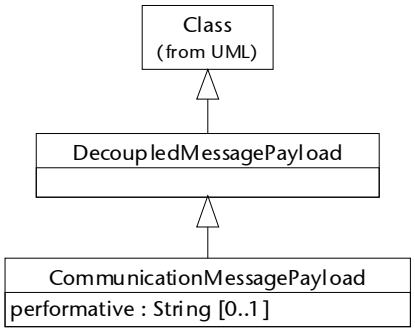


Fig. 5-12 Communicative Interactions—payloads

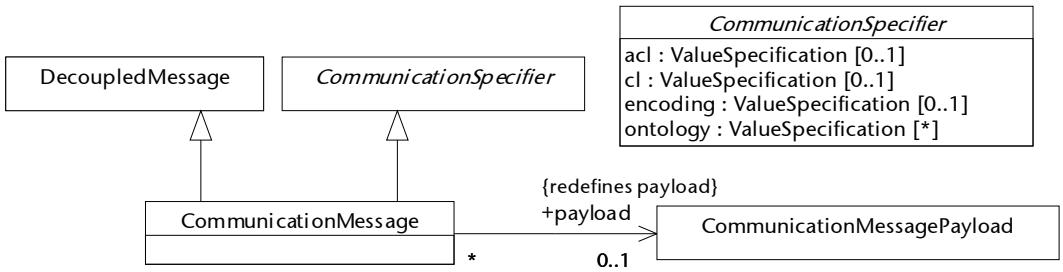


Fig. 5-13 Communicative Interactions—communication specifier and communication message

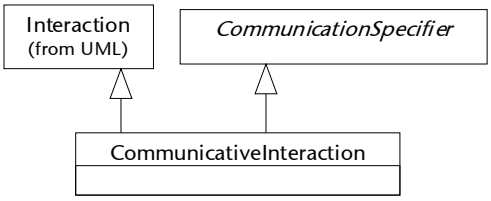


Fig. 5-14 Communicative Interactions—communicative interaction

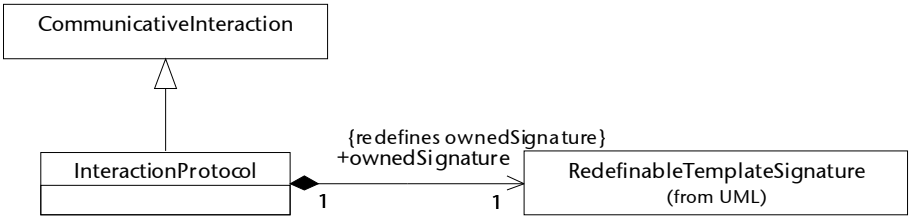


Fig. 5-15 Communicative Interactions—interaction protocol

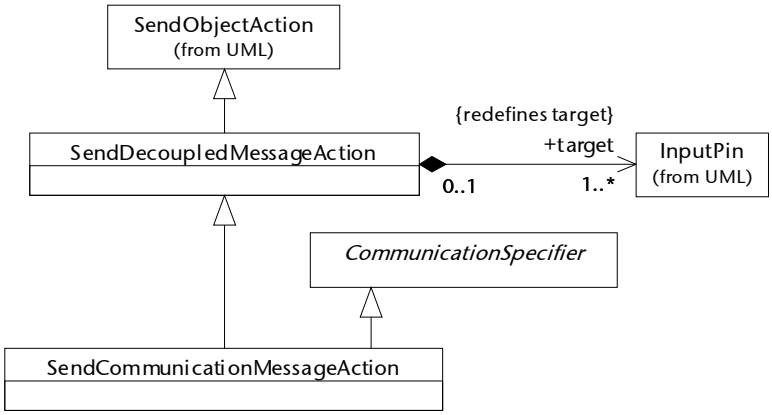


Fig. 5-16 Communicative Interactions—send message actions

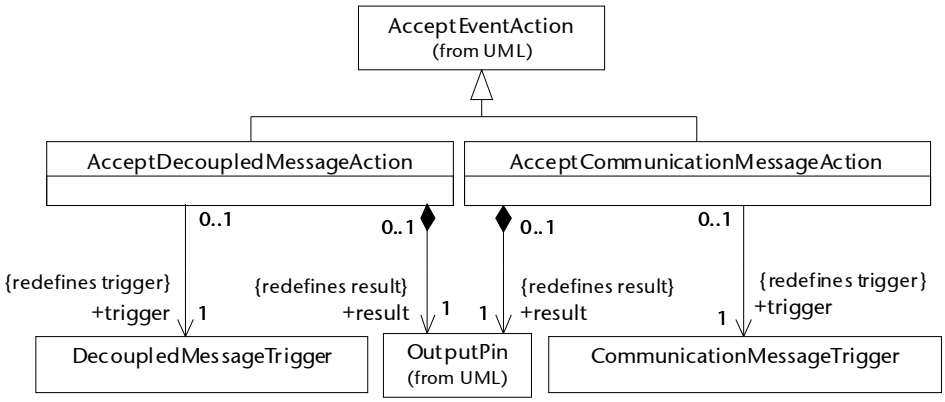


Fig. 5-17 Communicative Interactions—accept message actions

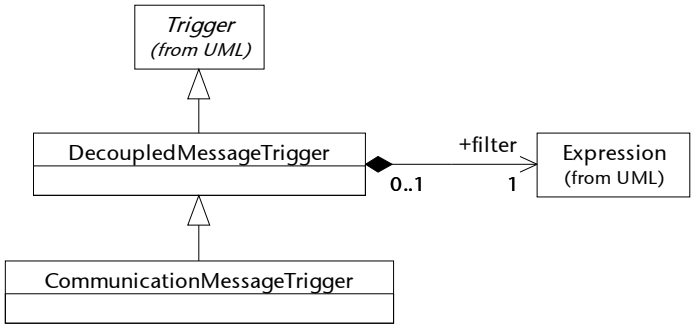


Fig. 5-18 Communicative Interactions—message triggers

5.3.1 MultiLifeline

Semantics MultiLifeline is a specialized Lifeline (from UML) and MultiplicityElement (from UML), used to represent a multivalued ConnectableElement (i.e. ConnectableElement with multiplicity > 1) participating in an Interaction (from UML). The multiplicity meta-attribute of the MultiLifeline determines the number of instances it represents. If the multiplicity is equal to 1, MultiLifeline is semantically identical with Lifeline (from UML).

The selector of a MultiLifeline may (in contrary to Lifeline) specify more than one participant represented by the MultiLifeline.

Notation MultiLifeline is depicted by a Lifeline symbol having specified the multiplicity mark. Information identifying the MultiLifeline has following format:

lifelineident ::= [*connectable_element_name* ['[' *selector* ']'] [':'*class_name* '['*multiplicity*']'] [*decomposition*] | 'self' '['*multiplicity*']'

Syntax and semantics of *connectable_element_name*, *selector*, *class_name*, *decomposition*, and 'self' is defined in UML Lifeline. Syntax and semantics of the *multiplicity* is given by UML MultiplicityElement. See Fig. 5-19.

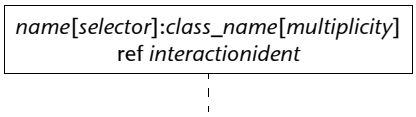


Fig. 5-19 Notation of MultiLifeline

The same extension of Lifeline's notation (including the presentation options) applies also for Lifelines in Communication Diagrams.

Presentation options The multiplicity for a MultiLifeline may also be shown as a multiplicity mark in the top right corner of the "head" box, see Fig. 5-20.

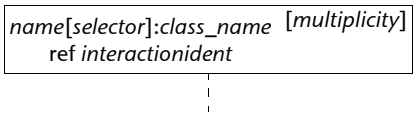


Fig. 5-20 Alternative notation of MultiLifeline

Examples Fig. 5-21 presents a usage of MultiLifelines in a specification of the FIPA Contract Net Protocol (see [24] for details).

The initiator issues call for proposal (cfp) messages to m participants. The participants generate n responses (n<=m). Within the participants two subgroups can be identified: n-j refusers who refuse the cfp and j proponents who propose to perform the task, by sending the propose message (j<=n). Once the deadline passes, the initiator evaluates the received j proposals and selects agents to perform the task. The k (k<=j) rejected agents will be sent a reject-proposal message and the remaining j-k accepted proponents will receive an accept-proposal message. Once each single accepted proponent has completed the task, it sends a completion message to the initiator in the form of an inform-done or an inform-result act. If the task completion fails, a failure message is sent back to the initiator.

Rationale MultiLifeline is introduced to represent a multivalued ConnectableElement participating in an Interaction.

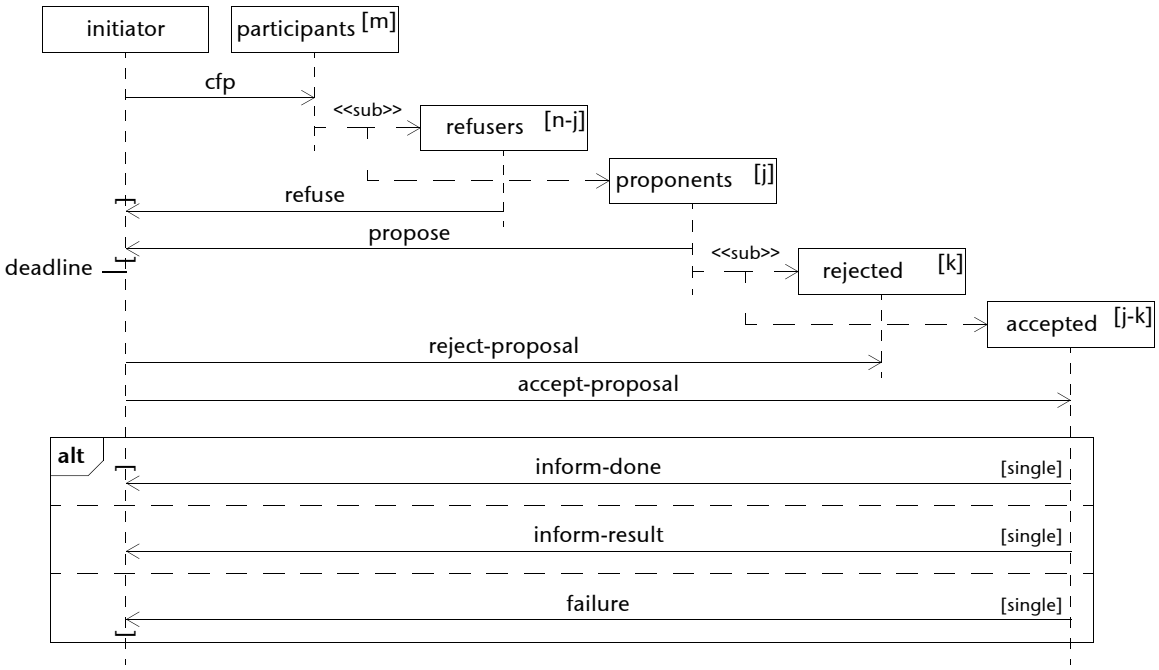


Fig. 5-21 FIPA Contract Net Protocol

5.3.2 MultiMessage

Semantics MultiMessage is a specialized Message (from UML) which is used to model a particular communication between MultiLifelines (p. 70) of an Interaction.

If the sender of a MultiMessage is a MultiLifeline, the MultiMessage represents a set of messages of a specified kind sent from all instances (potentially constrained by the sendDiscriminator) represented by that MultiLifeline.

If the receiver of a MultiMessage is a MultiLifeline, the MultiMessage represents a set of messages of a specified kind multicasted to all instances (potentially constrained by the receiveDiscriminator) represented by that MultiLifeline.

If a message end of a MultiMessage references a simple Lifeline (from UML), it represents a single sender or receiver.

When a sender and/or receiver of a MultiMessage are represented by MultiLifelines, the owned constraints sendDiscriminator and receiveDiscriminator can be used to specify what particular representatives of the group of ConnectableElements represented by the particular MultiLifeline are involved in the communication modeled by that MultiMessage.

Within an alternative CombinedFragment (from UML), it is useful to differentiate between:

- *all* of the ConnectableElements represented by the MultiLifeline, and
- *each* of the ConnectableElements represented by the MultiLifeline.

The keyword 'single' used as the corresponding discriminator indicating the latter of the above cases.

The receiver of a MultiMessage can be a group of instances containing also the senders themselves. In this case the MultiMessage can specify (by the toItself meta-attribute) whether the message is sent also to the senders themselves or not.

Attributes

toItself: Boolean[1]	If *true*, the MultiMessage is sent also to its sender when the sender belongs to the group of receivers. If *false*, the sender is excluded from the group of receivers.

Associations

sendDiscriminator: Constraint[0..1]	The Constraint which specifies the subset of MultiMessage senders when it is sent from a MultiLifeline. Senders are those instances represented by the MultiLifeline for which the sendDiscriminator is evaluated to *true*. Subsets UML Element::ownedElement.
receiveDiscriminator: Constraint[0..1]	The Constraint which specifies the MultiMessage receivers when it is sent to a MultiLifeline. Receivers are those instances represented by the MultiLifeline for which the receiveDiscriminator is evaluated to *true*. Subsets UML Element::ownedElement.

Constraints

1. At least one end of the MultiMessage must be a MultiLifeline:

   self.sendEvent.covered.oclIsKindOf(MultiLifeline) or
   self.receiveEvent.covered.oclIsKindOf(MultiLifeline)

2. The sendDiscriminator meta-association can be specified only if the sender is represented by a MultiLifeline:

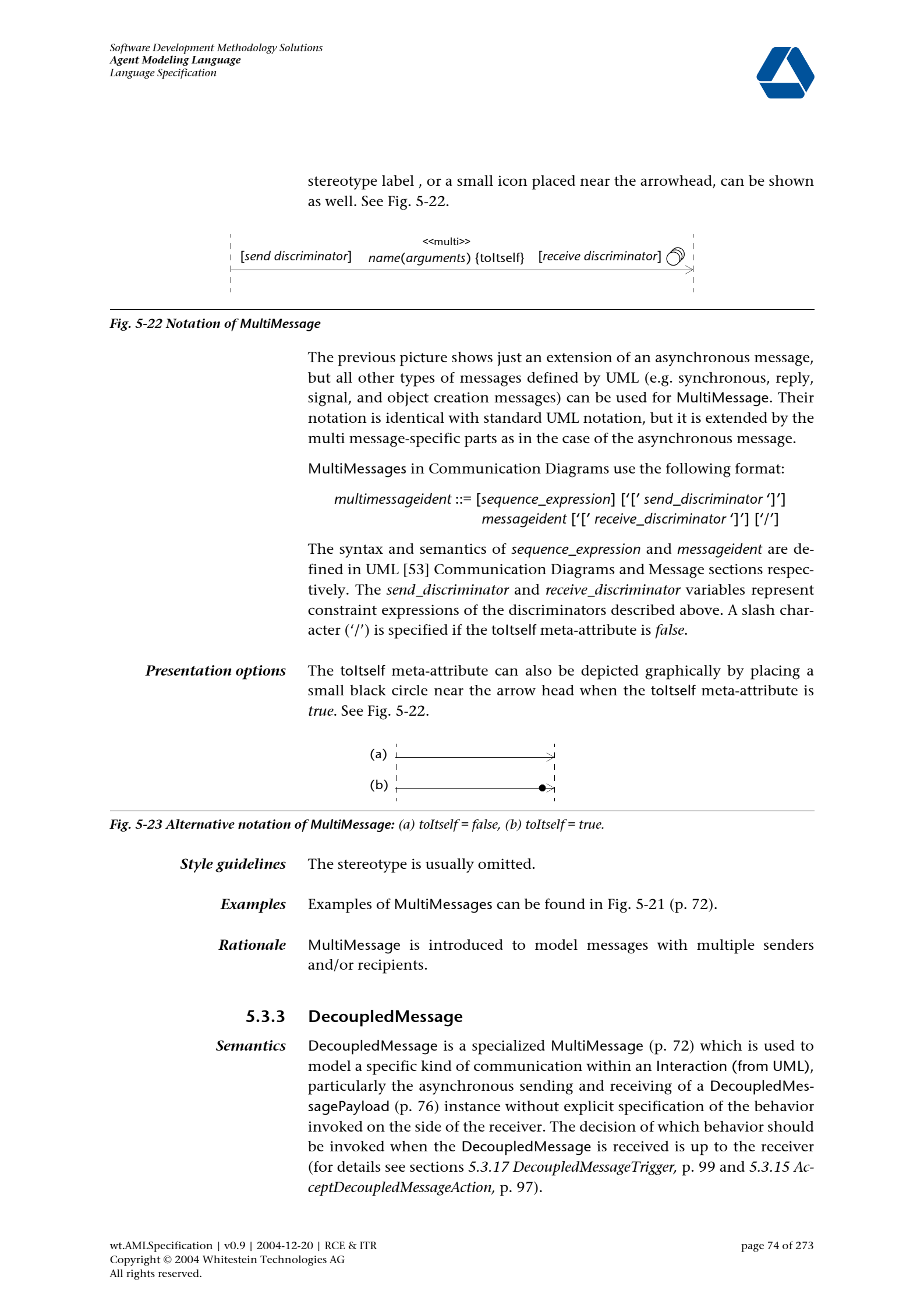
   self.sendDiscriminator->notEmpty() implies
   self.sendEvent.covered.oclIsKindOf(MultiLifeline)

3. The receiveDiscriminator meta-association can be specified only if the receiver is represented by a MultiLifeline:

   self.receiveDiscriminator->notEmpty() implies
   self.receiveEvent.covered.oclIsKindOf(MultiLifeline)

Notation

MultiMessage is depicted as a UML Message having the discriminator labels placed within brackets near the message ends and the keyword toItself, shown as a property string, placed near the name, if it is *true*. The <<multi>>



stereotype label , or a small icon placed near the arrowhead, can be shown as well. See Fig. 5-22.

Fig. 5-22 Notation of MultiMessage

The previous picture shows just an extension of an asynchronous message, but all other types of messages defined by UML (e.g. synchronous, reply, signal, and object creation messages) can be used for MultiMessage. Their notation is identical with standard UML notation, but it is extended by the multi message-specific parts as in the case of the asynchronous message.

MultiMessages in Communication Diagrams use the following format:

multimessageident ::= [*sequence_expression*] ['[' *send_discriminator* ']'] *messageident* ['[' *receive_discriminator* ']'] ['/']

The syntax and semantics of *sequence_expression* and *messageident* are defined in UML [53] Communication Diagrams and Message sections respectively. The *send_discriminator* and *receive_discriminator* variables represent constraint expressions of the discriminators described above. A slash character ('/') is specified if the toItself meta-attribute is *false*.

Presentation options The toItself meta-attribute can also be depicted graphically by placing a small black circle near the arrow head when the toItself meta-attribute is *true*. See Fig. 5-22.

Fig. 5-23 Alternative notation of MultiMessage: *(a) toItself = false, (b) toItself = true.*

Style guidelines The stereotype is usually omitted.

Examples Examples of MultiMessages can be found in Fig. 5-21 (p. 72).

Rationale MultiMessage is introduced to model messages with multiple senders and/or recipients.

5.3.3 DecoupledMessage

Semantics DecoupledMessage is a specialized MultiMessage (p. 72) which is used to model a specific kind of communication within an Interaction (from UML), particularly the asynchronous sending and receiving of a DecoupledMessagePayload (p. 76) instance without explicit specification of the behavior invoked on the side of the receiver. The decision of which behavior should be invoked when the DecoupledMessage is received is up to the receiver (for details see sections *5.3.17 DecoupledMessageTrigger,* p. 99 and *5.3.15 AcceptDecoupledMessageAction,* p. 97).

The objects transmitted in the form of DecoupledMessages are DecoupledMessagePayload (p. 76) instances.

Because all the decoupled messages are asynchronous, the messageSort meta-attribute (inherited from the UML Message) is ignored.

Associations

payload: DecoupledMessagePayload[0..1]	The type of the object transmitted.

Constraints

1. The constraints [2], [3], and [4] imposed on the UML Message are released, i.e. the DecoupledMessage's signature does not need to refer to either an Operation or a Signal.

Notation

DecoupledMessage is shown as an asynchronous MultiMessage, but the stereotype label is <<decoupled>>, or a different small icon is placed near the arrowhead. See Fig. 5-24.

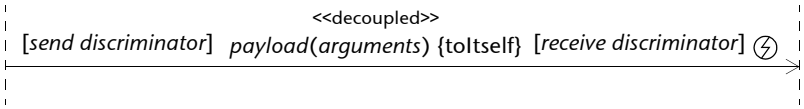


Fig. 5-24 Notation of DecoupledMessage

Syntax for the DecoupledMessage name is the following:

dm-messageident ::= *payload*['(' *arguments* ')']

If the meta-attribute payload is specified, the DecoupledMessage's signature must refer to the DecoupledMessagePayload, i.e.:

- the *payload* must correspond to the referred DecoupledMessagePayload's name, and
- the *arguments* use the same notation as defined for UML Message, and must correspond to the attributes of the DecoupledMessagePayload (i.e. the name of an argument must be the same as the name of one of the attributes of the referenced DecoupledMessagePayload, and the type of the argument must be of the same kind as the type of the attribute with the corresponding name).

If the meta-attribute payload is not specified, the *payload* can be any string and the *attributes* can be any values.

Presentation options

As for MultiMessage.

Style guidelines

The stereotype is usually omitted.

Argument names may by omitted if mapping of argument values to the payload attributes is unambiguous.

Examples

An example can be obtained by the replacement of all MultiMessages in Fig. 5-21 (p. 72) by DecoupledMessages.

For another example see Fig. 5-26.

Rationale DecoupledMessage is introduced to model autonomy in message processing.

5.3.4 DecoupledMessagePayload

Semantics DecoupledMessagePayload is a specialized Class (from UML) used to model the type of objects transmitted in the form of DecoupledMessages (p. 74).

Notation DecoupledMessagePayload is depicted as a UML Class with the stereotype <<dm payload>> and/or a special icon, see Fig. 5-25.

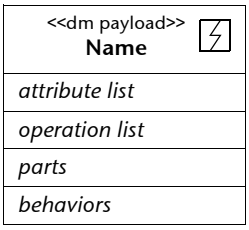


Fig. 5-25 Notation of DecoupledMessagePayload

Style guidelines The DecoupledMessagePayload usually specifies only attributes, and possibly also corresponding access operations.

Examples Fig. 5-26 (a) shows an example of DecoupledMessagePayload named SendOrder. Its instances represent the payloads of the DecoupledMessages used to send an order from broker to the stock exchange.

Example of such a DecoupledMessage is depicted in Fig. 5-26 (b).

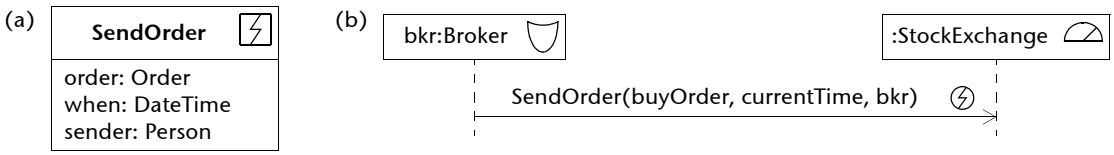


Fig. 5-26 Example of DecoupledMessagePayload

Rationale DecoupledMessagePayload is introduced to model objects transmitted in the form of DecoupledMessages.

5.3.5 Subset

Semantics Subset is a specialized Dependency (from UML) used to specify that instances represented by one Lifeline are a subset of instances represented by another Lifeline. The Subset relationship is between:

- an EventOccurrence owned by the “superset” Lifeline (client), and
- the “subset” Lifelines (suppliers),

used to specify that since the occurrence of the supersetEvent, some of the instances represented by the “superset” Lifeline are represented also by the “subset” Lifeline.

The “subset” Lifeline’s selector (for the details about the selector see Lifeline [53] and section *5.3.1 MultiLifeline,* p. 70) specifies the instances of the “superset” Lifeline that are also represented by the “subset” Lifeline.

All instances represented by the “subset” Lifeline are still represented also by the “superset” Lifeline.

One Lifeline can represent a “subset” of several “superset” Lifelines, i.e. more than one Subset relationships can lead to one “subset” Lifeline.

Termination of the “subset” Lifeline (the Stop is placed at the end of Lifeline) destroys all instances it represents.

Associations

supersetEvent: EventOccurrence[1]	An EventOccurrence owned by the “superset” Lifeline. It specifies the time point when the subset of represented instances is identified. Redefines UML Dependency::client.
subset: Lifeline[1..*]	The “subset” Lifeline. Redefines UML Dependency::supplier.

Constraints

1. All types of the “subset” Lifelines must conform to the type of the “superset” Lifeline:

   ```
   self.subset->forAll((represents.type->notEmpty() and
   self.supersetEvent.covered.represents.type->notEmpty()) implies
   self.subset.represents.type.conformsTo(
   self.supersetEvent.covered.represents.type))
   ```

Notation

The Subset relationship is depicted as a UML Dependency with the stereotype <<sub>>. The dependency arrowhead is always connected to the “subset” Lifeline “head”. See Fig. 5-27 and Fig. 5-28.

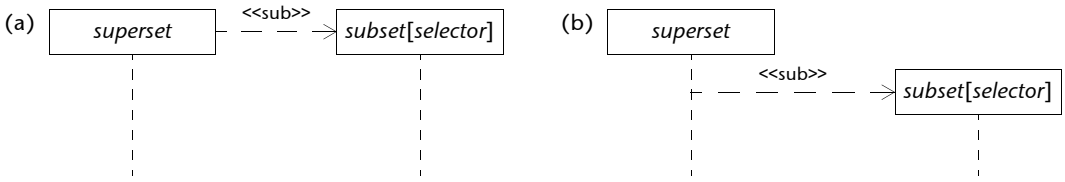


Fig. 5-27 Notation of Subset: *(a) the subset is identified from the beginning of the superset’s existence, (b) the subset is identified during the life time of the superset.*

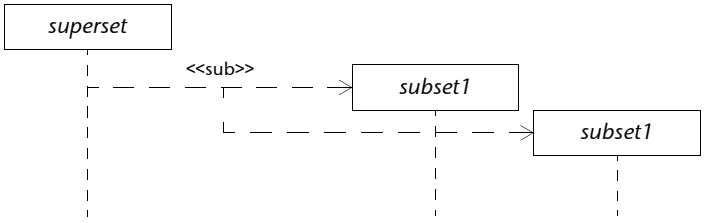


Fig. 5-28 Notation of Subset—multiple subsets created at once

When shown in a Communication Diagram, the Subset relationship can also specify a sequence-expression in order to identify a relative time since which the subset Lifeline has been identified. If the Subset relationship does not specify a sequence-expression, the subset Lifeline exists from the time when the superset Lifeline exists in the interaction. See Fig. 5-29.

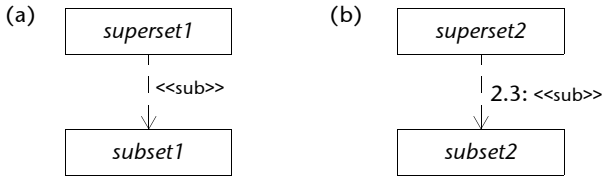


Fig. 5-29 Notation of Subset in Communication Diagram: *(a) subset1 starts to be identified from when superset1 occurs, (b) subset2 is identified when an "event" 2.3 occurs during Interaction.*

Style guidelines The name is usually omitted.

Examples Examples of Subset relationship can be found in Fig. 5-21 (p. 72), and Fig. 5-32 (p. 80).

Rationale Subset is introduced to specify that instances represented by one Lifeline are a subset of instances represented by another Lifeline.

5.3.6 Join

Semantics Join is a specialized Dependency (from UML) used to specify joining of instances represented by one Lifeline with a set of instances represented by another Lifeline. The Join relationship is between:

- ❒ an EventOccurrence owned by a "subset" Lifeline (client) and
- ❒ an EventOccurrence owned by a "union" Lifeline (supplier),

having a possibility to specify a selector constraint on the "subset" Lifeline.

It is used to specify that a subset of instances, which have been until the subsetEvent represented by the "subset" Lifeline, is after the unionEvent represented only by the "union" Lifeline. The "union" Lifeline, thus after the unionEvent represents the union of the instances it has been representing before, and the instances specified by the Join dependency.

The subset of instances of the "subset" Lifeline joining the "union" Lifeline is given by the AND-ed combination of the Join's selector and the selector of the "union" Lifeline.

If the selector of the Join dependency is not specified, all the instances represented by the “subset” Lifeline conforming to the “union” Lifeline’s selector are joined.

Between subsetEvent and unionEvent occurrences, the set of instances joining the “union” Lifeline is not represented by any of the two Lifelines.

One EventOccurrence can be a client or a supplier of several Joins.

Associations

selector: Expression [0..1]	Specifies the subset of instances represented by the “subset” Lifeline, which are being joined. Subsets UML Element::ownedElement.
subsetEvent: EventOccurrence[1]	An EventOccurrence owned by the subset Lifeline. It specifies the time point when the subset of instances resented by the “subset” Lifeline is detached from others. Redefines UML Dependency::client.
unionEvent: EventOccurrence[1]	An EventOccurrence owned by the “union” Lifeline. It specifies the time point when the subset instances are joined with the union set of instances. Redefines UML Dependency::supplier.

Constraints

1. The Lifeline owning the EventOccurrence referred to by the unionEvent meta-association must be a MultiLifeline:

   self.unionEvent.covered.oclIsKindOf(MultiLifeline)

2. The type of the subsetEvent’s Lifeline must conform to the type of the unionEvent’s MultiLifeline:

   (self.subsetEvent.covered.represents.type->notEmpty() and
   self.unionEvent.covered.represents.type->notEmpty()) implies
   self.subsetEvent.covered.represents.type.conformsTo(
   self.unionEvent.covered.represents.type)

Notation

The Join relationship is depicted as UML Dependency with the stereotype <<join>>. The dependency arrowhead is always connected to the subset Lifeline “head”. Optionally a selector expression can be specified in brackets. See Fig. 5-30 and Fig. 5-28.

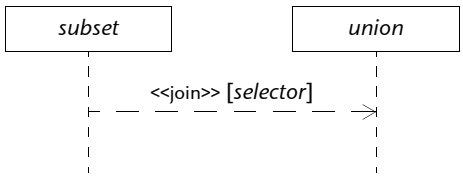


Fig. 5-30 Notation of Join

When shown in a Communication Diagram, the Join relationship can optionally specify a sequence-expression in order to identify a relative time when it occurs. See Fig. 5-31.

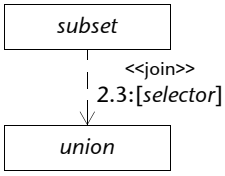


Fig. 5-31 Notation of Join in Communication Diagrams

Style guidelines The name is usually omitted.

Examples Fig. 5-33 shows an interaction representing a communication of parties concerned with the processing of a job application, which exploits the Join relationship.

The Applicant sends a message applyForJob to a Company. The Company replies with an invitation message that specifies all the details about the interview. The interview is described by a separate interaction between the Applicant and a StaffManager. If the applicant is accepted after the Interview, she/he becomes an employee (joins a group of employees) and the entity role apl is disposed (but the instance which was playing that entity role still exist).

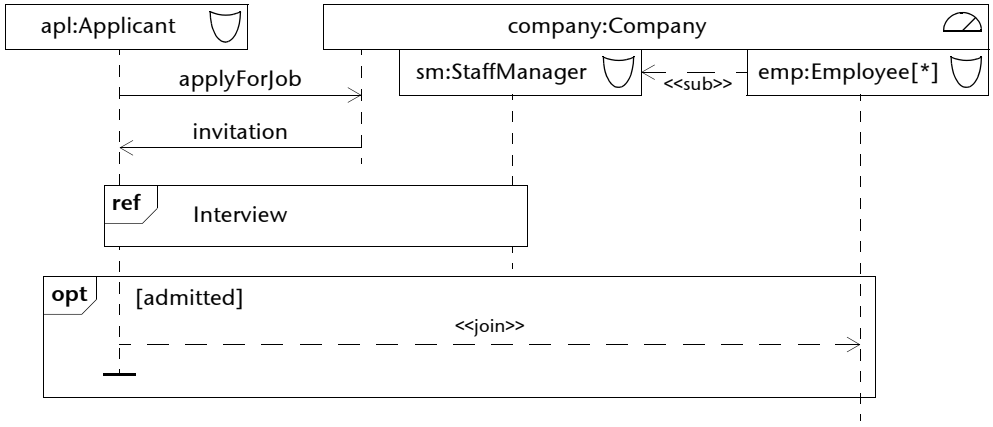


Fig. 5-32 Example of Join

Fig. 5-33 shows a part of an interaction representing a situation when an actively playing soccer player is substituted by another player during the match.

A player selected from a group of actively (currently) playing players joins a group of passive players placed outside the soccer pitch. Then a selected passive player becomes active.

Rationale Join is introduced to specify the joining of instances represented by one Lifeline with a set of instances represented by another Lifeline.

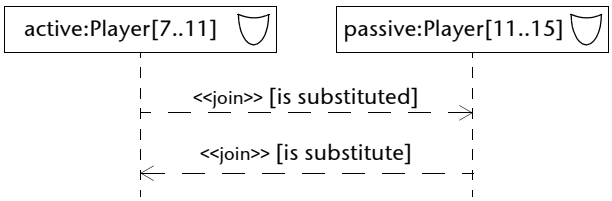


Fig. 5-33 Example of a Join with selector

5.3.7 AttributeChange

Semantics

AttributeChange is a specialized InteractionFragment (from UML) used to model the change of attribute values (state) of ConnectableElements (from UML) in the context of Interactions (from UML).

AttributeChange enables adding and removing attribute values in time, as well as to express added attribute values by Lifelines (from UML). Attributes are represented by inner ConnectableElements.

AttributeChange can also be used to model dynamic changing of entity roles played by behavioral entities represented by Lifelines. Furthermore, it allows the modeling of entity interaction with respect to the played entity roles, i.e. each “sub-lifeline” representing a played entity role (or entity roles in the case of MultiLifeline, p. 70) is used to model the interaction of its player with respect to this/these entity role(s).

If an AttributeChange is used to destroy played entity roles, it represents disposal of the entity roles while their former players still exist as instances in the system. To also destroy the player of an entity role, the Stop element (from UML) must be used instead. Usage of the Stop element thus leads to the disposal of the player as well as all the entity roles it has been playing.

Associations

createdLifeline: Lifeline[*]	A set of Lifelines representing the added attribute values. Subsets UML InteractionFragment::covered.
destroyedLifeline: Lifeline[*]	A set of lifelines representing the removed attribute values. Subsets UML InteractionFragment::covered.
owningLifeline: Lifeline[0..1]	The Lifeline representing an owner of the created attribute values.
when: EventOccurrence[1]	The EventOccurrence specifying a time point when the AttributeChange occurs.

Constraints

1. If createdLifeline is specified, the owningLifeline must be specified as well:

   self.createdLifeline->notEmpty() implies self.owningLifeline->notEmpty()

2. Each createdLifeline must represent an attribute of the Classifier used as the type of the ConnectableElement represented by the owningLifeline meta-association:

   (self.createdLifeline->notEmpty() and
   self.owningLifeline.represents.type->notEmpty()) implies
   self.owningLifeline.represents.type.attribute->
   includesAll(self.createdLifeline.represents)

Notation AttributeChange is depicted as a bold horizontal line. The created Lifelines are shown as Lifelines with the top sides of their "heads" attached to the AttributeChange line. The owner of new Lifelines is identified by drawing a small solid circle at the intersection of its Lifeline and AttributeChange's line. If necessary, the AttributeChange's line is enlarged to touch the owner's Lifeline. Vertical lines of Lifelines representing the destroyed attribute values terminate at the AttributeChange's line. Lifelines that's vertical lines cross the AttributeChange's line and continue below are unaffected by the AttributeChange. See Fig. 5-34.

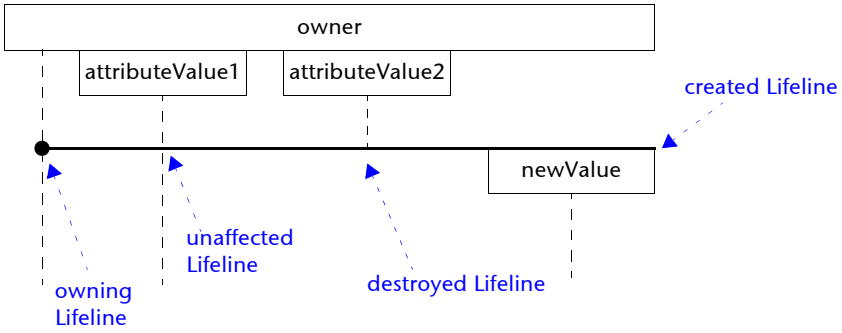


Fig. 5-34 Notation of AttributeChange

In Communication Diagrams, the AttributeChange is shown as a small solid circle placed into a Lifeline representing the owner of newly created or destroyed Lifelines. A sequence number can be placed near the circle, to identify the relative time at which the AttributeChange occurs. Destroyed Lifelines are identified by arrows with bold crossing bars as arrow heads and lead from the AttributeChange circle. Created Lifelines are identified by arrows with open arrow heads leading from the AttributeChange circle. Fig. 5-35 shows a Communication Diagram semantically identical to the Sequence Diagram shown in Fig. 5-34.

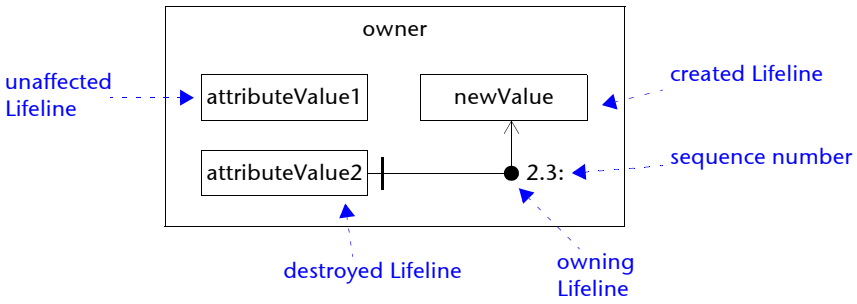


Fig. 5-35 Notation of AttributeChange in Communication Diagrams

Style guidelines The name is usually omitted.

Examples The change of an entity role played by one entity is shown in Fig. 5-36. An agent worker is a programmer. After receiving a message advancement from the ProjectBoard it changes its entity role to the projectManager. At this point he is no longer a programmer. The argument values of the advancement message specify details of advancement (including the type of a new entity role) but for simplicity they are hidden in the diagram.

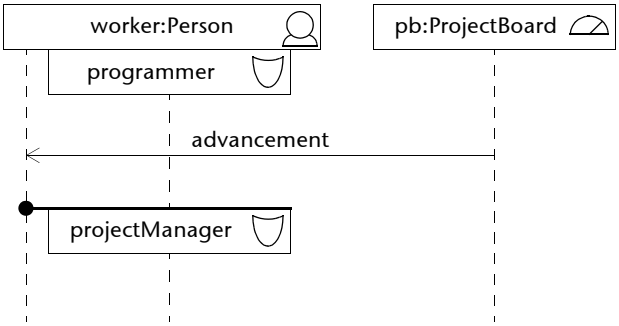


Fig. 5-36 Example of AttributeChange—entity role change

An addition of a new entity role to the entity is shown in Fig. 5-37. An agent worker is an analyst. At a certain time the projectManager decides to also allocate this worker to testing. He informs the worker about this decision by sending a takeResponsibility message. The worker then also becomes a tester, but still remains an analyst. The argument values of the takeResponsibility message specify details of the new entity role being played but for simplicity they are hidden in the diagram.

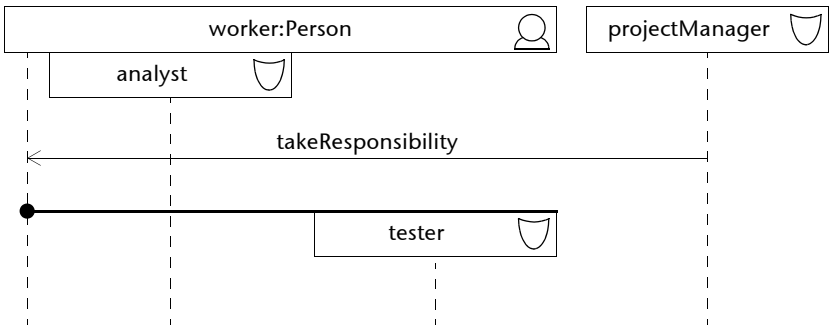


Fig. 5-37 Example of AttributeChange—entity role addition

Change of the player of an entity role is shown in Fig. 5-38. An agent worker is a programmer. After receiving a message advancement from the ProjectBoard it changes its entity role to the projectManager. At this point he is no longer a programmer. The argument values of the advancement message specify details of the advancement (including the type of a new entity role) but for simplicity they are hidden in the diagram.

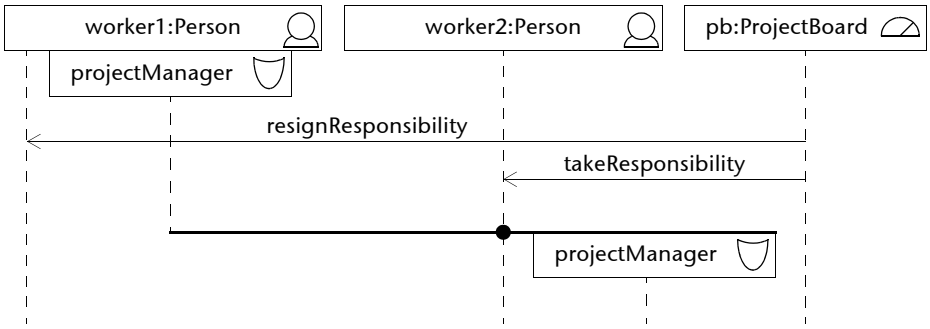


Fig. 5-38 Example of AttributeChange—change of entity role players

Fig. 5-39 shows the difference between destroying the entity role's Lifeline by the AttributeChange and stopping the Lifeline by the Stop element. The diagram shows two employees, John and Robert. After John betrayed the confidential company information he is fired and is no longer an employee. But he still exists. The AttributeChange was used in this case. The owning Lifeline does not need to be specified because no Lifeline was created.

On the other hand, after a fatal injury Robert dies, i.e. Robert as a person no longer exists. The Stop was used in this case.

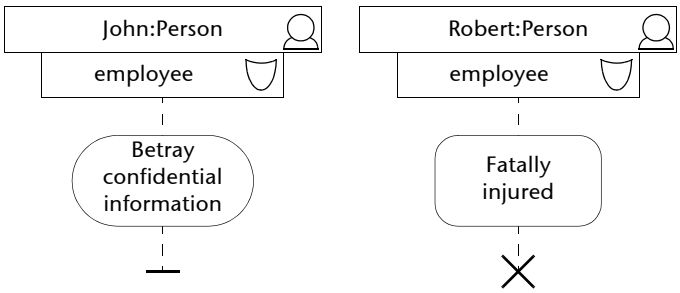


Fig. 5-39 Example of destruction and stopping of an entity role's Lifeline

Rationale AttributeChange is introduced to model a change of the attribute values (state) of ConnectableElements in the context of Interactions.

5.3.8 CommunicationSpecifier

Semantics CommunicationSpecifier is an abstract metaclass which defines meta-properties of its concrete subclasses, i.e. CommunicationMessage (p. 86), CommunicativeInteraction (p. 89), and ServiceSpecification (p. 103), which are used to model different aspects of communicative interactions.

CommunicationMessages can occur in CommunicativeInteractions, and parametrized CommunicativeInteractions can be parts of ServiceSpecifications. All of them can specify values of the meta-attributes inherited from the CommunicationSpecifier. Potential conflicts in specifications of the CommunicationSpecifier's meta-property values are resolved by the overriding principle that defines which concrete subclasses of the CommunicationSpecifier have higher priority in specification of those meta-attributes. Thus, if specified on different priority levels, the values at higher priority levels override those specified at lower priority levels.

The priorities, from the highest to the lowest are defined as follows:

1. CommunicationMessage,
2. CommunicativeInteraction,
3. ServiceSpecification.

For example, if an encoding value is specified for a particular CommunicationMessage it overrides the encoding specified for the owning CommunicativeInteraction. If the encoding is not specified for the CommunicationMessage, its value is specified by the owning CommunicativeInteraction. If not specified for the owning CommunicativeInteraction and the CommunicativeInteraction is a part of a ServiceSpecification, the value is taken from that ServiceSpecification. If the encoding is not specified even in the ServiceSpecification, it remains unspecified in the model.

Attributes

acl: ValueSpecification [0..1]	Denotes the *agent communication language* in which CommunicationMessages are expressed.

cl: ValueSpecification [0..1]	Denotes the language in which the CommunicationMessage's content is expressed, also called the *content language*.
encoding: ValueSpecification [0..1]	Denotes the specific encoding of the CommunicationMessage's content.
ontology: ValueSpecification[*]	Denotes the ontologies used to give a meaning to the symbols in the CommunicationMessage's content expression.

Notation There is no general notation for CommunicationSpecifier. Its specific subclasses define their own notation.

The CommunicationSpecifier's meta-attributes are specified as tagged values. The following keywords are used:

- acl,
- cl,
- encoding, and
- ontology.

Their values represent arbitrary ValueSpecifications, but the most commonly used types are: enumerations, string literals, or lists of literals (used for the ontology specification).

Style guidelines Usually, the CommunicationSpecifier's meta-attributes are specified as a hidden information, not shown in diagrams.

Rationale CommunicationSpecifier is introduced to define meta-properties which are used to model different aspects of communicative interactions. It is used in definitions of its subclasses.

5.3.9 CommunicationMessage

Semantics CommunicationMessage is a specialized DecoupledMessage (p. 74) and CommunicationSpecifier (p. 85), which is used to model communicative acts of *speech act based communication* in the context of Interactions.

The objects transmitted in the form of CommunicationMessages are CommunicationMessagePayload (p. 87) instances.

Associations

payload: CommunicationMessagePayload[0..1]	The type of the object transmitted. Redefines DecoupledMessage::payload.

Notation CommunicationMessage is shown as a DecoupledMessage (p. 74), but the stereotype label is <<communication>>, or a different small icon is placed near the arrowhead. See Fig. 5-40.

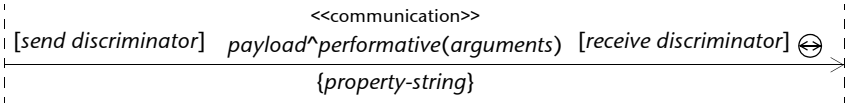


Fig. 5-40 Notation of CommunicationMessage

Syntax for the CommunicationMessage name is the following:

cm-messageident ::= [*payload* '^'] *performative*['(' *arguments* ')']

If the meta-attribute payload is specified, the CommunicationMessage's signature must refer to the CommunicationMessagePayload, i.e.:

- the *payload* (optional) must correspond to the referred CommunicationMessagePayload's name,
- the *performative* must correspond to the performative of the CommunicationMessagePayload, and
- the *arguments* use the same notation as defined for UML Message, and must correspond (for details see section *5.3.3 DecoupledMessage,* p. 74) to the attributes of the CommunicationMessagePayload.

If the meta-attribute payload is not specified, the *performative* can be any string and the *attributes* can be any values. In this case the *payload* part is omitted from the signature.

Presentation options As for MultiMessage (p. 72).

Style guidelines When used in a CommunicativeInteraction (p. 89) containing only CommunicationMessages, their stereotypes are usually omitted.

Argument names may by omitted if mapping of argument values to the payload attributes is unambiguous.

Usually, the specification of tagged values is a hidden information, not shown in diagrams.

Examples See Fig. 5-42 (p. 88) and Fig. 5-48 (p. 92).

Rationale CommunicationMessage is introduced to model speech act based communication in the context of Interactions.

5.3.10 CommunicationMessagePayload

Semantics CommunicationMessagePayload is a specialized Class (from UML) used to model the type of objects transmitted in the form of CommunicationMessages (p. 86).

Attributes

performative: String [0..1]	Performative of the CommunicationMessagePayload.

Notation CommunicationMessagePayload is depicted as a UML Class with the stereotype <<cm payload>> and/or a special icon, see Fig. 5-41.

If specified, the value of the meta-attribute performative is depicted as a property string (tagged value) with name 'performative', placed in the name compartment.

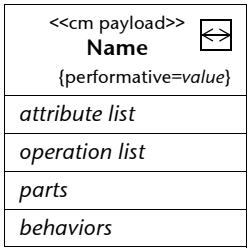


Fig. 5-41 Notation of CommunicationMessagePayload

Presentation options If the only displayed tagged value is the performative, its keyword may be omitted and only its value is specified.

Style guidelines The CommunicationMessagePayload usually specifies only performative, attributes, and possibly also corresponding access operations.

Examples Fig. 5-42 (a) shows an example of a CommunicationMessagePayload named PerformTask. Its instances represent the payloads of the CommunicationMessages used to send the request to perform a task.

Examples of such CommunicationMessages are depicted in Fig. 5-26 (b). Presented notational alternatives are semantically equivalent (for details see section *5.3.9 CommunicationMessage,* p. 86).

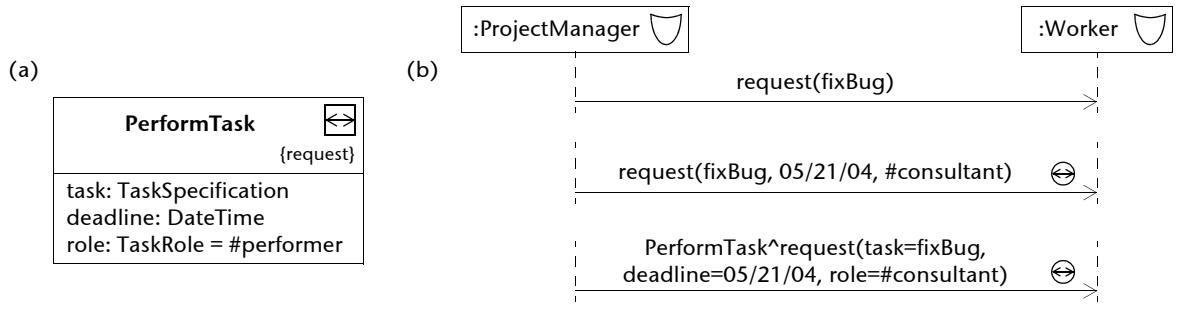


Fig. 5-42 Example of CommunicationMessagePayload

Rationale CommunicationMessagePayload is introduced to model objects transmitted in the form of CommunicationMessages.

5.3.11 CommunicativeInteraction

Semantics CommunicativeInteraction is a specialized Interaction (from UML) and CommunicationSpecifier (p. 85), used to model speech act based communications, i.e. Interactions containing CommunicationMessages (p. 86).

CommunicativeInteraction, being a concrete subclass of the abstract CommunicationSpecifier, can specify some additional meta-attributes of interactions, which are not allowed to be specified within UML Interactions, particularly:

- acl, i.e. the agent communication language used within the CommunicativeInteraction,
- cl, i.e. the content language used within the CommunicativeInteraction,
- encoding, i.e. the content encoding used within the CommunicativeInteraction, and
- ontology, i.e. the ontologies used within the CommunicativeInteraction.

For the above meta-attributes, the overriding principle defined in section *5.3.8 CommunicationSpecifier* (p. 85) holds.

Notation CommunicativeInteraction is depicted as UML Interaction with the optionally specified meta-attribute tagged values (shown as a property string) placed into the name compartment of the diagram frame. See Fig. 5-43.

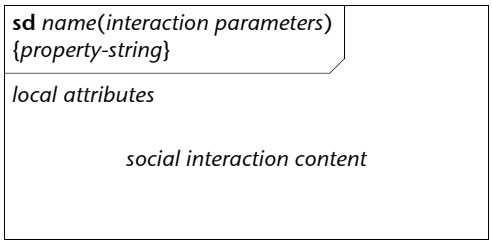


Fig. 5-43 Notation of CommunicativeInteraction

Examples See Fig. 5-48 (p. 92).

Rationale CommunicativeInteraction is introduced to model speech act based communications.

5.3.12 InteractionProtocol

Semantics InteractionProtocol is a parametrized CommunicativeInteraction (p. 89) template used to model reusable templates of CommunicativeInteractions.

Possible TemplateParameters of an InteractionProtocol are:

- values of CommunicationSpecifier's meta-attributes,
- local variable names, types, and default values,
- Lifeline names, types, and selectors,
- Message names and argument values,

- MultiLifeline (p. 70) multiplicities,
- MultiMessage (p. 72) discriminators,
- CommunicationMessage (p. 86) meta-attributes,
- ExecutionOccurrence's behavior specification,
- guard expressions of InteractionOperands,
- specification of included Constraints, and
- included Expressions and their particular operands.

Partial binding of an InteractionProtocol (i.e. the TemplateBinding which does not substitute all the template parameters by actual parameters) results in a different InteractionProtocol.

A complete binding of an InteractionProtocol represents a CommunicativeInteraction.

Associations

ownedSignature: RedefinableTemplateSignature[1]	A template signature specifying the formal template parameters. Redefines UML Classifier::ownedSignature.

Notation InteractionProtocol is depicted as a parametrized CommunicativeInteraction, with the keyword 'ip' used for the diagram frame kind. Specification of template parameters is placed into a dashed rectangle in the upper right corner of the diagram frame. See Fig. 5-43.

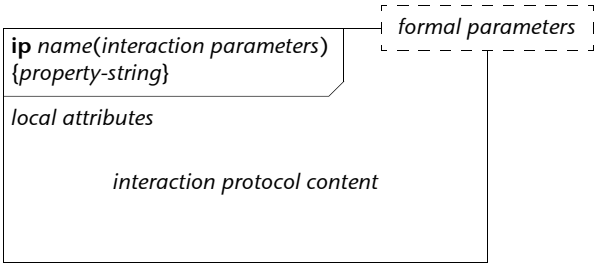


Fig. 5-44 Notation of InteractionProtocol

Binding of an InteractionProtocol is shown either as a TemplateBinding relationship, or named bound CommunicativeInteraction, or as an anonymous bound CommunicativeInteraction, see Fig. 5-45. All notational variants are defined in UML 2.0 Superstructure [53].

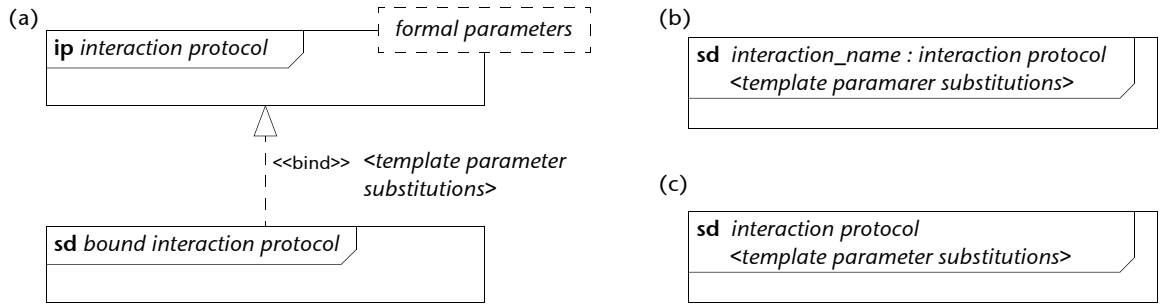


Fig. 5-45 Notation of InteractionProtocol binding: *(a) as a TemplateBinding relationship, (b) as a named bound CommunicativeInteraction, (c) as an anonymous bound CommunicativeInteraction.*

Presentation options Binding of a formal template parameter representing a type used within the InteractionProtocol (e.g. a type of comprised Lifeline) can be alternatively shown as a dashed line from the bound InteractionProtocol to the Type substituting the formal parameter. The line is labelled with a name of the substituted template parameter. See Fig. 5-46.

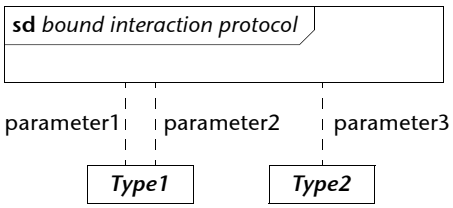


Fig. 5-46 Alternative notation of InteractionProtocol binding

A partially bound InteractionProtocol can show remaining unbound (free) template parameters explicitly in a dashed rectangle placed in the upper right corner of the diagram frame, see Fig. 5-47.

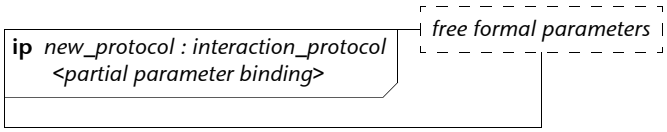


Fig. 5-47 Alternative notation of partially bound InteractionProtocol

Examples Fig. 5-48 shows the FIPA Request Interaction Protocol [24] modeled as InteractionProtocol.

The initiator requests the participant to perform some action by sending a request CommunicationMessage. The participant processes the request and makes a decision whether to accept or refuse it. If a refuse decision is taken, the participant communicates a refuse CommunicationMessage and both entity roles are destroyed. Otherwise the interaction continues.

If conditions indicate that an explicit agreement is required (that is, “notification necessary” is *true*), the participant communicates an agree CommunicationMessage. Once the request has been agreed upon and the action has been completed, the participant must communicate the action result

as either failure, or inform-done, or inform-result CommunicationMessage. After the interaction has finished, both entity roles are destroyed.

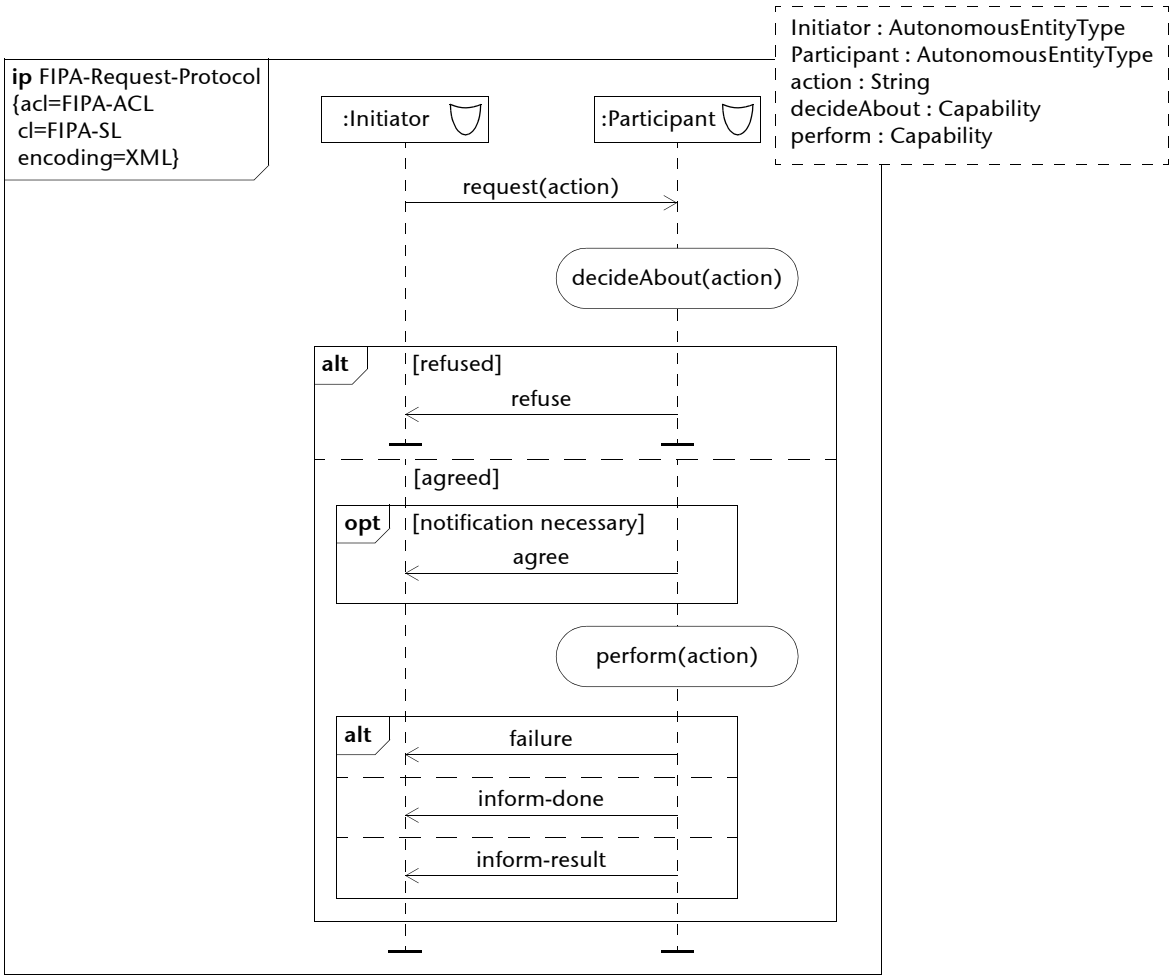


Fig. 5-48 Example of InteractionProtocol depicted as a Sequence Diagram

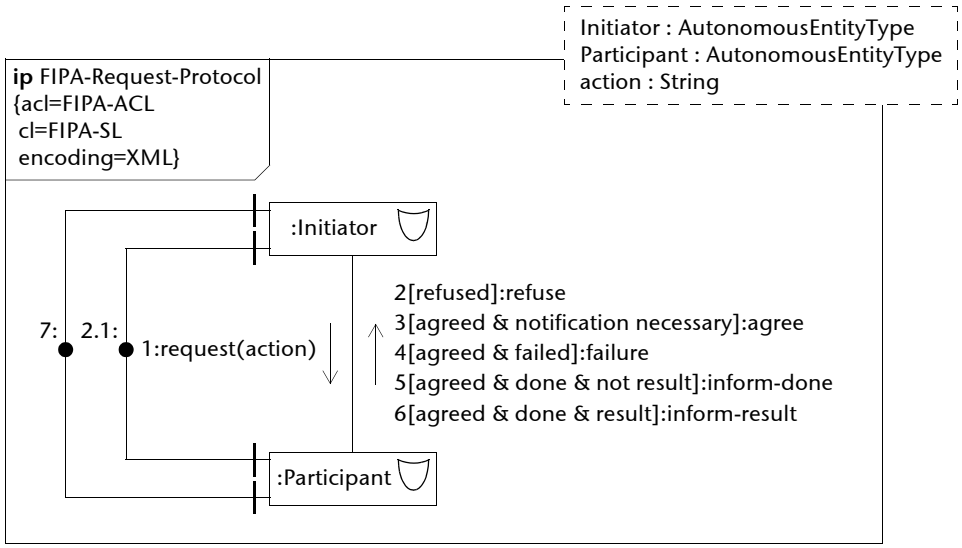


Fig. 5-49 shows a simplified version of the previously defined interaction protocol FIPA-Request-Protocol in the form of Communication Diagram.

Fig. 5-49 Example of InteractionProtocol depicted as a Communication Diagram

Fig. 5-50 shows binding of the previously defined interaction protocol FIPA-Request-Protocol. The result of the binding is a new CommunicativeInteraction called BuyRequest, used to model an interaction of a personal assistant agent with a portfolio manager agent when the personal assistant requests the portfolio manager to buy some securities.

The initiator template parameter is substituted by the PersonalAssistant AgentType, participant by the PortfolioManager AgentType, action by a string specifying the buying action, decideAbout by the buyingAssessment operation, and the parameter perform by the do operation. The diagram also shows that the SocialAssociation between the PersonalAssistant and the PortfolioManager is realized by the BuyRequest CommunicativeInteraction which specifies this relationship at the detailed level of abstraction.

Rationale InteractionProtocol is introduced to model reusable templates of CommunicativeInteractions.

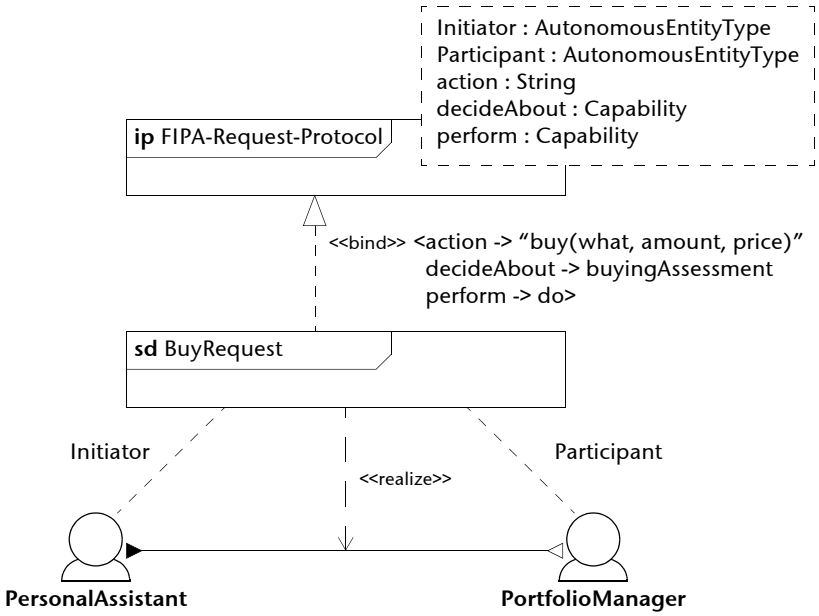


Fig. 5-50 Example of InteractionProtocol binding

5.3.13 SendDecoupledMessageAction

Semantics SendDecoupledMessageAction is a specialized SendObjectAction (from UML) used to model the action of sending of DecoupledMessagePayload (p. 76) instances, referred to by the request meta-association, in the form of a DecoupledMessage (p. 74) to its recipient(s), referred to by the target meta-association.

Associations

target: InputPin[1..*]	The target objects to which the DecoupledMessage is sent. Redefines UML SendObjectAction::target.

Notation SendDecoupledMessageAction is depicted as a rectangle with convex rounded right side, see Fig. 5-51.

Syntax of the SendDecoupledMessageAction name is the same as defined for the *dm-messageident* by DecoupledMessage:

dm-messageident ::= *payload*['(' *arguments* ')']

The *payload* represents the type of the sent DecoupledMessagePayload instance, and the *arguments* can be used to indicate the values of the attributes of the sent payload instance. If the type of the InputPin referred to by the request meta-association is specified, the *payload* must correspond to the name, and *arguments* must correspond to its attributes of the type.

Recipients of the sent DecoupledMessage are indicated by a comma separated list of recipient names placed below the message name in parentheses.

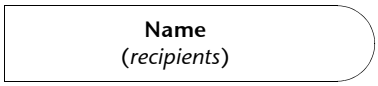


Fig. 5-51 Notation of SendDecoupledMessageAction

Presentation options If the action's name does not conform to the syntax of the *dm-messageident*, the *dm-messageident* may optionally be placed after the list of recipients separated by the double colon. See Fig. 5-52.

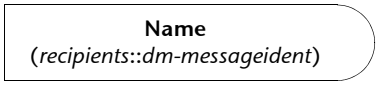


Fig. 5-52 Alternative notation of SendDecoupledMessageAction—indication of recipients and sent decoupled message

SendDecoupledMessageAction can be alternatively depicted as a UML Action with stereotype <<send decoupled message>> and/or a special decoration icon, see Fig. 5-53.

Fig. 5-53 Alternative notation of SendDecoupledMessageAction—action with stereotype and/or decoration icon

Examples See Fig. 5-57 (p. 97). All SendCommunicationMessageActions from the diagram should be replaced by SendDecoupledMessageActions.

Rationale SendDecoupledMessageAction is introduced to model the sending of DecoupledMessages in Activities.

5.3.14 SendCommunicationMessageAction

Semantics SendCommunicationMessageAction is a specialized SendDecoupledMessageAction (p. 94), which allows to specify the values of the CommunicationSpecifier's (p. 85) meta-attributes.

Notation SendCommunicationMessageAction is depicted as a rectangle with convex rounded right side and doubled left side, see Fig. 5-54.

Syntax of the SendCommunicationMessageAction name is the same as defined for the *cm-messageident* by CommunicationMessage:

cm-messageident ::= [*payload* '^'] *performative*['(' *arguments* ')']

The *payload* represents the name, the *performative* denotes the performative, and the *arguments* the values of the attributes of the type of the sent CommunicationMessagePayload instance. If the type of the InputPin referred to by the request meta-association is specified, the *payload* must correspond to its name, the *performative* to its performative meta-attribute, and *arguments* must correspond to its attributes.

Recipients of the sent CommunicationMessage are indicated by the comma separated list of recipient names, placed below the message name in parentheses.

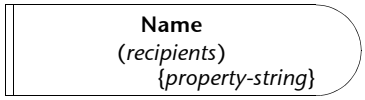


Fig. 5-54 Notation of SendCommunicationMessageAction

If specified, the values of the CommunicationSpecifier's meta-attributes are depicted as a property string (tagged values).

Presentation options

If the action's name does not conform to the syntax of the *cm-messageident*, the *cm-messageident* may optionally be placed after the list of recipients separated by the double colon. See Fig. 5-55.

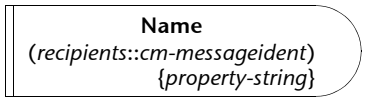


Fig. 5-55 Alternative notation of SendCommunicationMessageAction—indication of recipients and sent communication message

SendCommunicationMessageAction can be alternatively depicted as a UML Action with stereotype <<send communication message>> and/or a special decoration icon, see Fig. 5-56.

Fig. 5-56 Alternative notation of SendCommunicationMessageAction—action with stereotype and/or decoration icon

Examples

An activity diagram of participant's behavior within the FIPA Contract Net Protocol (see [24] for details) is depicted in Fig. 5-57. The participant waits until a cfp CommunicationMessage arrives. If so, the cfp is evaluated, and based on this evaluation the cfp is either refused, and the refuse CommunicationMessage is sent to the initiator, or accepted. In the case of acceptance, the bid is computed and proposed to the initiator by sending the propose CommunicationMessage. Then the initiator sends either the reject-proposal, which terminates the algorithm, or the accept-proposal. In the second case the participant performs the action and informs the initiator of the result of the action execution, by sending the inform-result CommunicationMessage.

The diagram does not show recipients of CommunicationMessages because the only recipient is the initiator of the FIPA Contract Net Protocol.

For more examples see also Fig. 5-120 (p. 133) and Fig. 5-122 (p. 136).

Rationale

SendCommunicationMessageAction is introduced to model the sending of CommunicationMessages in Activities.

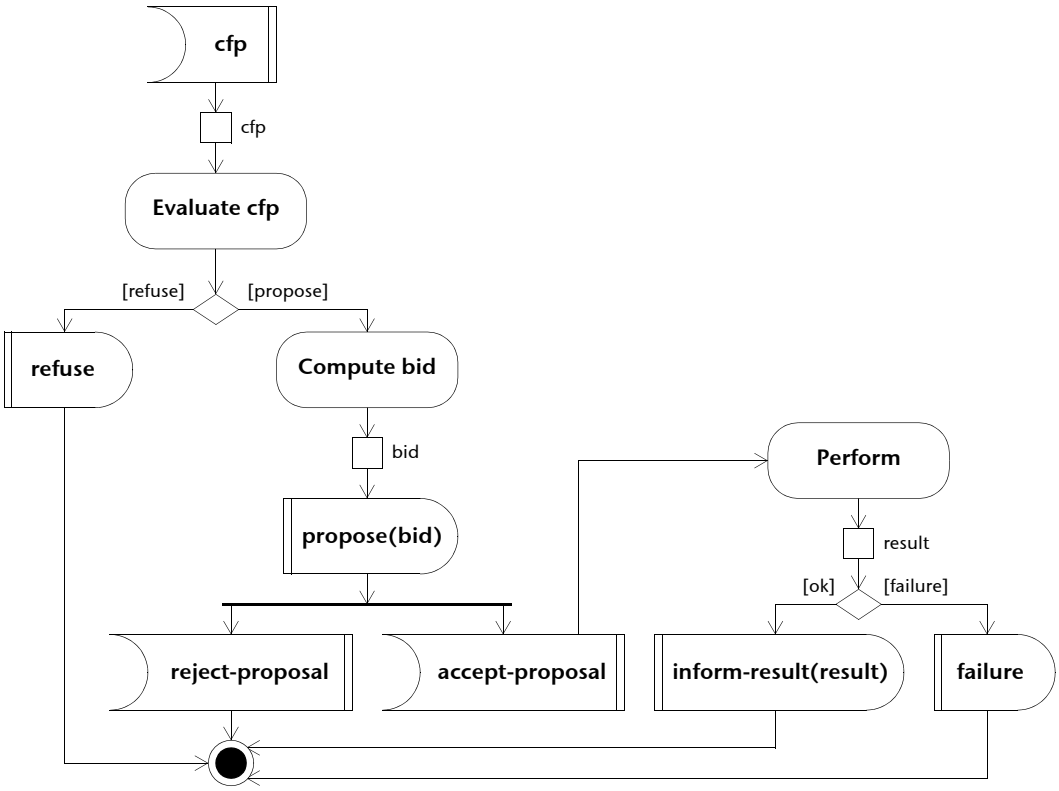


Fig. 5-57 Example of SendCommunicationMessageAction and AcceptCommunicationMessageAction

5.3.15 AcceptDecoupledMessageAction

Semantics

AcceptDecoupledMessageAction is a specialized AcceptEventAction (from UML) which waits for the reception of a DecoupledMessage (p. 74) that meets conditions specified by the associated trigger (for details see section *5.3.17 DecoupledMessageTrigger,* p. 99). The received DecoupledMessagePayload (p. 76) instance is placed to the result OutputPin.

If an AcceptDecoupledMessageAction has no incoming edges, the action starts when the containing Activity (from UML) or StructuredActivityNode (from UML) starts. An AcceptDecoupledMessageAction with no incoming edges is always enabled to accept events regardless of how many are accepted. It does not terminate after accepting an event and outputting the value, but continues to wait for subsequent events.

Associations

trigger: DecoupledMessageTrigger[1]	The DecoupledMessageTrigger accepted by the action. Redefines UML AcceptEventAction::trigger.
result: OutputPin[1]	The OutputPin holding the event object that has been received as a DecoupledMessage. Redefines UML AcceptEventAction::result.

Constraints

1. If the type of the OutputPin referred to by the result meta-association is specified, it must be a DecoupledMessagePayload:

   self.result.type->notEmpty() implies
   self.result.type.oclIsKindOf(DecoupledMessagePayload)

Notation AcceptDecoupledMessageAction is depicted as a rectangle with concave rounded left side, see Fig. 5-58.

The name of the AcceptDecoupledMessageAction is the associated trigger.

The event object received as a DecoupledMessage may be specified as an OutputPin.

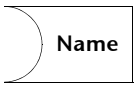


Fig. 5-58 Notation of AcceptDecoupledMessageAction

Presentation options AcceptDecoupledMessageAction can be alternatively depicted as a UML Action with stereotype <<accept decoupled message>> and/or a special decoration icon, see Fig. 5-59.

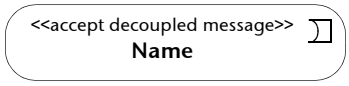


Fig. 5-59 Alternative notation of AcceptDecoupledMessageAction

Examples An example can be obtained by the replacement of all AcceptDecoupledMessageActions in Fig. 5-57 (p. 97) by AcceptCommunicationMessageActions.

Rationale AcceptDecoupledMessageAction is introduced to model the reception of DecoupledMessages in Activities.

5.3.16 AcceptCommunicationMessageAction

Semantics AcceptCommunicationMessageAction is a specialized AcceptEventAction (from UML) which waits for the reception of a CommunicationMessage (p. 86) that meets conditions specified by associated trigger (for details see section *5.3.18 CommunicationMessageTrigger,* p. 100). The received CommunicationMessagePayload (p. 87) instance is placed to the result OutputPin.

If an AcceptCommunicationMessageAction has no incoming edges, then the action starts when the containing Activity (from UML) or StructuredActivityNode (from UML) starts. An AcceptCommunicationMessageAction with no incoming edges is always enabled to accept events regardless of how many are accepted. It does not terminate after accepting an event and outputting a value, but continues to wait for subsequent events.

Associations

trigger: Communication-MessageTrigger[1]	The CommunicationMessageTrigger accepted by the action. Redefines UML AcceptEventAction::trigger.
result: OutputPin[1]	The OutputPin holding the event object that has been received as a CommunicationMessage. Redefines UML AcceptEventAction::result.

Constraints

1. If the type of the OutputPin referred to by the result meta-association is specified, it must be a CommunicationMessagePayload:

   self.result.type->notEmpty() implies
   self.result.type.oclIsKindOf(CommunicationMessagePayload)

Notation

AcceptCommunicationMessageAction is depicted as a rectangle with concave rounded left side and doubled right side, see Fig. 5-60.

The name of the AcceptCommunicationMessageAction is the associated trigger.

The event object received as a CommunicationMessage may be specified as an OutputPin.

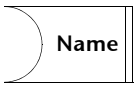


Fig. 5-60 Notation of AcceptCommunicationMessageAction

Presentation options

AcceptCommunicationMessageAction can be alternatively depicted as a UML Action with stereotype <<accept communication message>> and/or a special decoration icon, see Fig. 5-61.

Fig. 5-61 Alternative notation of AcceptCommunicationMessageAction

Examples

See Fig. 5-57 (p. 97), Fig. 5-120 (p. 133), and Fig. 5-122 (p. 136).

Rationale

AcceptCommunicationMessageAction is introduced to model the reception of CommunicationMessages in Activities.

5.3.17 DecoupledMessageTrigger

Semantics

DecoupledMessageTrigger is a specialized Trigger (from UML) that represents the event of reception of a DecoupledMessage (p. 74), that satisfies the condition specified by the boolean-valued Expression (from UML) referred to by the filter meta-association.

The Expression can constrain the signature name and argument values of the received DecoupledMessage, or alternatively, the type and attribute values of the received DecoupledMessagePayload (p. 76) instance.

Associations

filter: Expression[1]	A boolean-valued Expression filtering received DecoupledMessages.

Notation DecoupledMessageTrigger is denoted as a boolean-valued Expression which represents the value of the filter meta-attribute.

Style guidelines In the case of simple filtering, the following syntax of the DecoupledMessageTrigger can be used:

triggerident ::= *dm-messageident-list*
dm-messageident-list ::= *dm-messageident* [',' *dm-messageident-list*]

The *dm-messageident-list* represents a comma-separated list of *dm-messageidents* as defined by the DecoupledMessage. The DecoupledMessageTrigger accepts all DecoupledMessages that match the specified payload and argument values.

Examples The DecoupledMessageTrigger specified as:

CancelOrder, SendOrder(when=today)

accepts all DecoupledMessages CancelOrder, or SendOrder from today.

Rationale DecoupledMessageTrigger is introduced to model events representing reception of DecoupledMessages.

5.3.18 CommunicationMessageTrigger

Semantics CommunicationMessageTrigger is a specialized DecoupledMessageTrigger (p. 99) that represents the event of reception of a CommunicationMessage (p. 86), that satisfies the condition specified by the boolean-valued Expression (from UML) referred to by the filter meta-association.

The Expression can constrain the signature name and argument values of the received CommunicationMessage, or alternatively, the type, value of performative meta-attribute, and attribute values of the received CommunicationMessagePayload (p. 87) instance.

Notation The same as for DecoupledMessageTrigger.

Style guidelines In the case of simple filtering, the following syntax of the CommunicationMessageTrigger can be used:

triggerident ::= *cm-messageident-list*
cm-messageident-list ::= *cm-messageident* [',' *cm-messageident-list*]

The *cm-messageident-list* represents a comma-separated list CommunicationMessage names as defined by the CommunicationMessage. The CommunicationMessageTrigger accepts all CommunicationMessages that match the specified payload, performative and argument values.

Examples The CommunicationMessageTrigger specified as:

accept-proposal

accepts all CommunicationMessages with performative accept-proposal.

Rationale CommunicationMessageTrigger is introduced to model events representing reception of CommunicationMessages.

5.4 Services

Overview The *Services* package defines metaclasses used to model services, particularly their specification, provision and usage.

Abstract syntax The diagrams of the Services package are shown in figures Fig. 5-62 to Fig. 5-66.

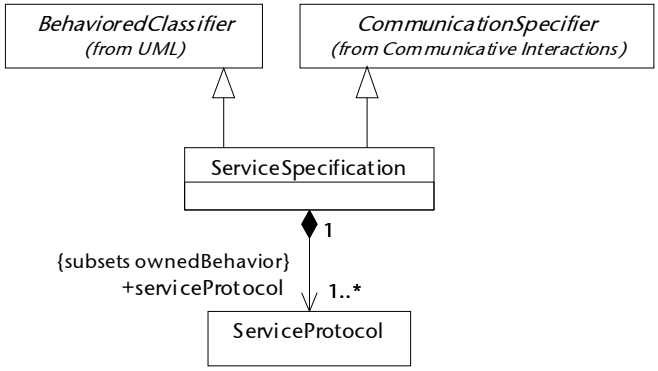


Fig. 5-62 Services—service specification

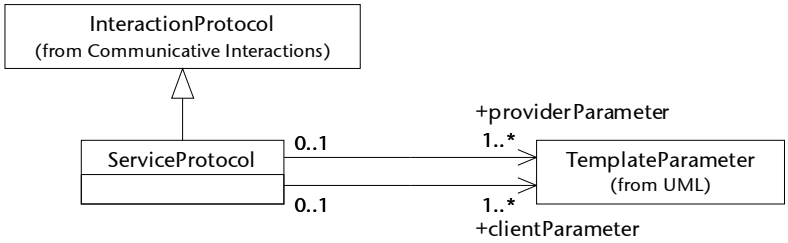


Fig. 5-63 Services—service protocol

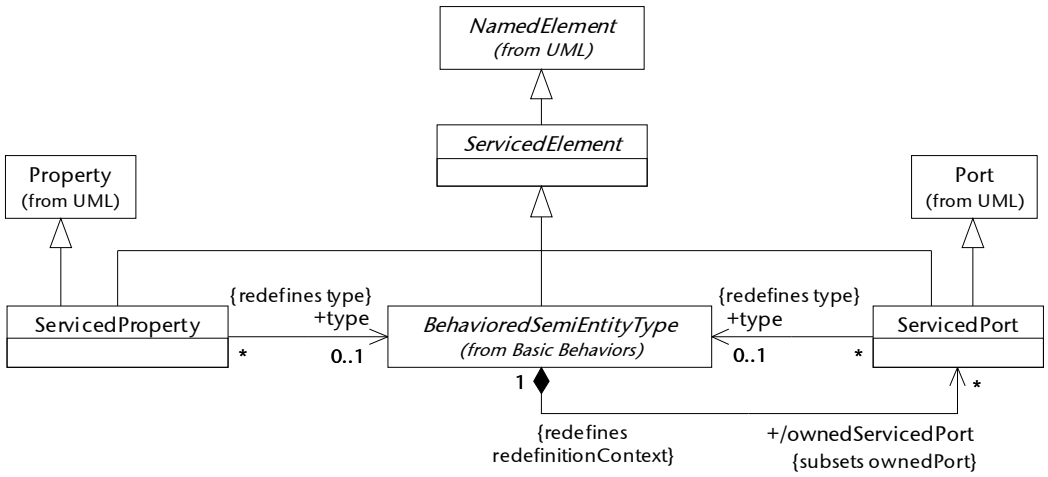


Fig. 5-64 Services—serviced elements

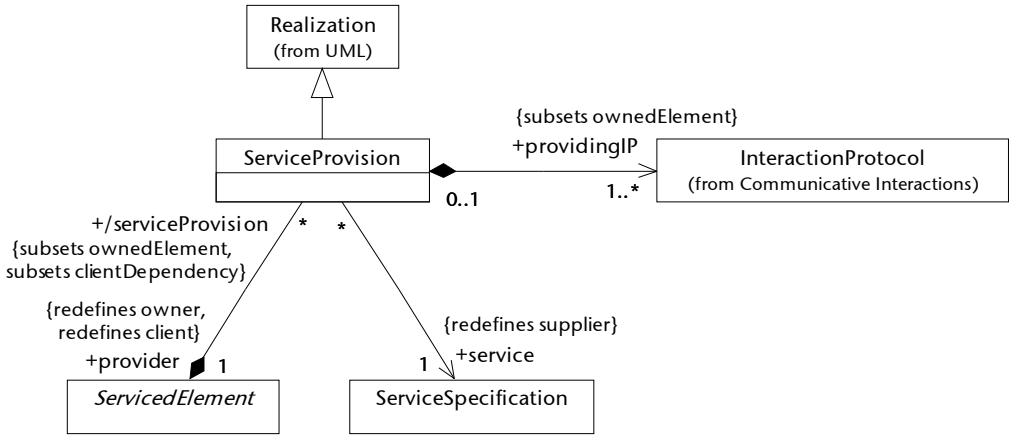


Fig. 5-65 Services—service provision

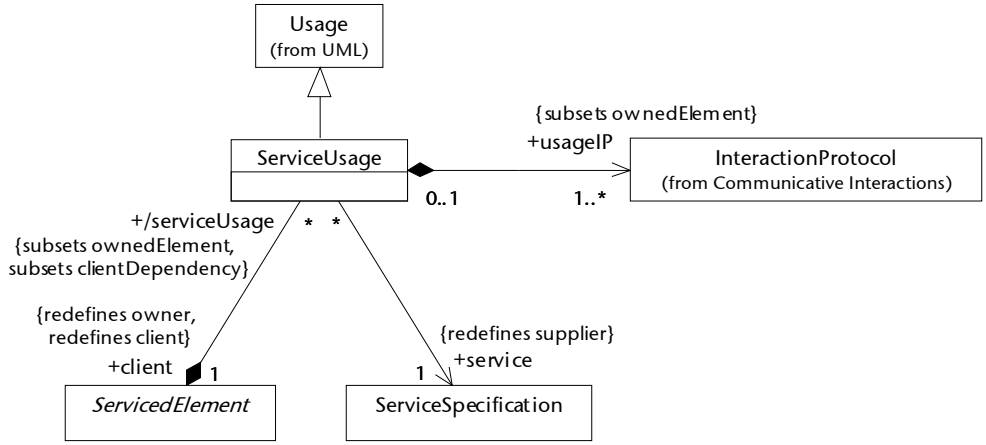


Fig. 5-66 Services—service usage

5.4.1 ServiceSpecification

Semantics

ServiceSpecification is a specialized BehavioredClassifier (from UML) and CommunicationSpecifier (p. 85), used to specify services.

A *service* is a coherent block of functionality provided by a behavioredsemi-entity, called *service provider*, that can be accessed by other behaviored semi-entities (which can be either external or internal parts of the service provider), called *service clients*. The ServiceSpecification is used to specify properties of such services, particularly:

- the functionality of the services and
- the way the specified service can be accessed.

The specification of the functionality and the accessibility of a service is modeled by owned ServiceProtocols (p. 105), i.e. InteractionProtocols (p. 89) extended with an ability to specify two mandatory, disjoint and nonempty sets of (not bound) parameters of their TemplateSignatures, particularly:

- provider template parameters, and
- client template parameters.

The *provider template parameters* (providerParameter meta-association) of all contained ServiceProtocols specify the set of template parameters that must be bound by the service providers, and the *client template parameters* (client-Parameter meta-association) of all contained ServiceProtocols specify the set of template parameters that must be bound by the service clients. Binding of all these complementary template parameters results in the specification of the CommunicativeInteractions between the service providers and the service clients.

For the meta-attributes defined by CommunicationSpecifier the overriding priority principle defined in *5.3.8 CommunicationSpecifier* (p. 85) applies.

Note 1: The ServiceSpecification can, in addition to the ServiceProtocols, also own other Behaviors (from UML) describing additional behavioral aspects of the service. For instance, *Interaction Overview Diagrams* (see UML for details) used to describe the overall algorithm (also called the process) of invoking particular ServiceProtocols.

Note 2: The ServiceSpecification can also contain StructuralFeatures to model additional structural characteristics of the service, e.g. attributes can be used to model the service parameters.

Associations

/serviceProtocol: ServiceProtocol[1..*]	Owned ServiceProtocols. This is a derived association. Subsets UML BehavioredClassifier::ownedBehavior.

Constraints

1. The serviceProtocol meta-association refers to all owned Behaviors of the kind ServiceProtocol:

```
serviceProtocol = self.ownedBehavior->
  select(oclIsKindOf(ServiceProtocol))
```

Notation ServiceSpecification is depicted as a UML Classifier symbol (i.e. a solid-outline rectangle containing the Classifier's name, and optionally also compartments separated by horizontal lines containing features or other members of the Classifier). A ServiceSpecification is marked with the stereotype <<service specification>> and/or a special decoration icon placed in the name compartment. The name compartment can also contain specification of tagged values, e.g. setting of meta-attributes e.g. inherited from the CommunicativeInteractionSpecifier metaclass.

The ServiceSpecification shows a compartment which comprises owned ServiceProtocols, but other standard UML or user-defined compartments can be used as well.

Notation of the ServiceSpecification is depicted in Fig. 5-67.

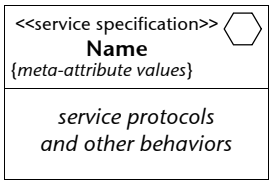


Fig. 5-67 Notation of the ServiceSpecification

Style guidelines When defined, the ServiceSpecification usually uses the rectangular notation with the compartment showing owned ServiceProtocols.

When used to specify provision or usage of the service, the ServiceSpecification is usually depicted just as an icon.

Examples Fig. 5-68 presents the specification of the FIPA-compliant DirectoryFacilitator [24] service with its functions: Register, Deregister, Modify, and Search. All functions use the Fipa-Request-Protocol, see [24] and Fig. 5-48 for details. Binding of the template parameters decideAbout and perform is omitted from this example, for the purpose of simplicity. The DirectoryFacilitator also specifies the acl, cl, and ontology meta-attributes used commonly in all comprised ServiceProtocols.

Rationale ServiceSpecification is introduced to model the specification of services, particularly (a) the functionality of the service, and (b) the way the service can be accessed.

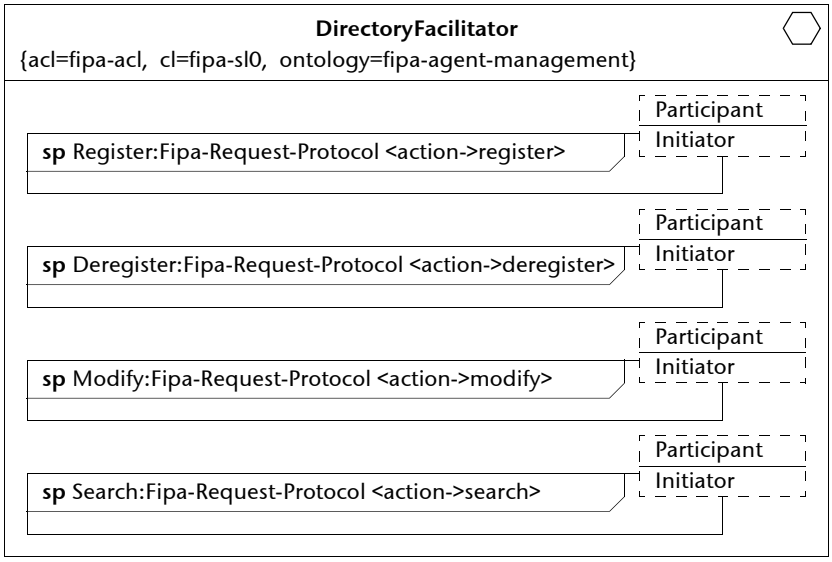


Fig. 5-68 Example of ServiceSpecification

5.4.2 ServiceProtocol

Semantics

ServiceProtocol is a specialized InteractionProtocol (p. 89), used only within the context of its owning ServiceSpecification (p. 103), extended with an ability to specify two mandatory, disjoint and non-empty sets of (not bound) parameters of its TemplateSignature (from UML), particularly:

- providerParameter, i.e. a set of parameters which must be bound by providers of the service, and
- clientParameter, i.e. a set of parameters which must be bound by clients of the service.

Usually at least one of the provider/client parameters is used as a Lifeline's type which represents a provider/client or its inner ConnectableElements (see UML StructuredClassifier).

The ServiceProtocol can be defined either as a unique InteractionProtocol (a parametrized CommunicativeInteraction) or as a partially bound, already defined InteractionProtocol.

Associations

providerParameter: TemplateParameter [1..*]	The set of TemplateParameters which must be bound by a provider of the service.
clientParameter: TemplateParameter [1..*]	The set of TemplateParameters which must be bound by a client of the service.

Constraints

1. The providerParameter refer only to the template parameters belonging to the signature owned by a ServiceProtocol:

self.ownedSignature.parameter->includesAll(providerParameter)

2. The clientParameter refers only to the template parameters belonging to the signature owned by a ServiceProtocol:

   self.ownedSignature.parameter->includesAll(clientParameter)

3. The providerParameter and clientParameter are disjoint:

   self.providerParameter->intersection(self.clientParameter)->isEmpty()

4. The providerParameter and clientParameter together cover all parameters of the template signature:

   self.providerParameter->union(self.clientParameter) =
   self.ownedSignature.paramater

Notation ServiceProtocol is shown as an InteractionProtocol, having a list of formal template parameters divided into two parts: provider parameters and client parameters. The provider parameters are preceded with the keyword <<provider>> and the client parameters with the keyword <<client>>. The keyword 'sp' used for the diagram frame kind. Notation of ServiceProtocol is depicted in Fig. 5-69.

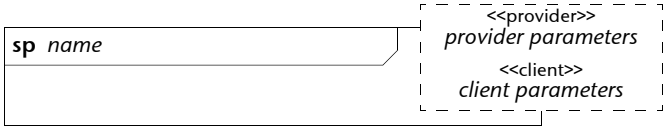


Fig. 5-69 Notation of ServiceProtocol

Presentation options Alternatively, the provider parameters can be separated from the client parameters by a solid horizontal line, see Fig. 5-70.

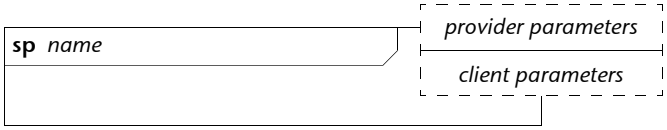


Fig. 5-70 Alternative notation of ServiceProtocol

Alternatively a ServiceProtocol can be shown textually with the following format:

service_protocol ::= *name* '/' *provider_parameters* '/' *client_parameters*

The *name* is the name of the ServiceProtocol, the *provider_parameters* is a comma-separated list of the provider parameters, and the *client_parameters* is a comma-separated list of the client parameters.

Examples Fig. 5-71 shows the FIPA Propose Interaction Protocol [24] modeled as ServiceProtocol.

The initiator sends a propose CommunicationMessage to the participant indicating that it will perform some action if the participant agrees. The participant responds by either rejecting or accepting the proposal, communi-

cating this with the reject-proposal or accept-proposal CommunicationMessage, accordingly.

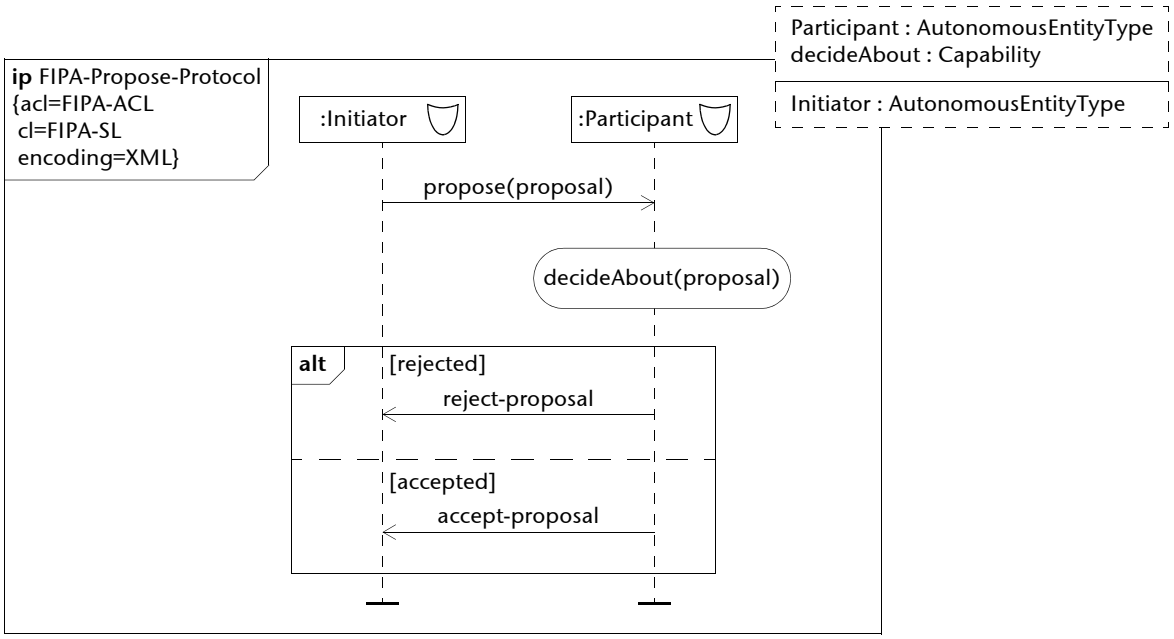


Fig. 5-71 Example of ServiceProtocol depicted as a Sequence Diagram

Fig. 5-72 shows a simplified version of the previously defined service protocol FIPA-Propose-Protocol in the form of Communication Diagram.

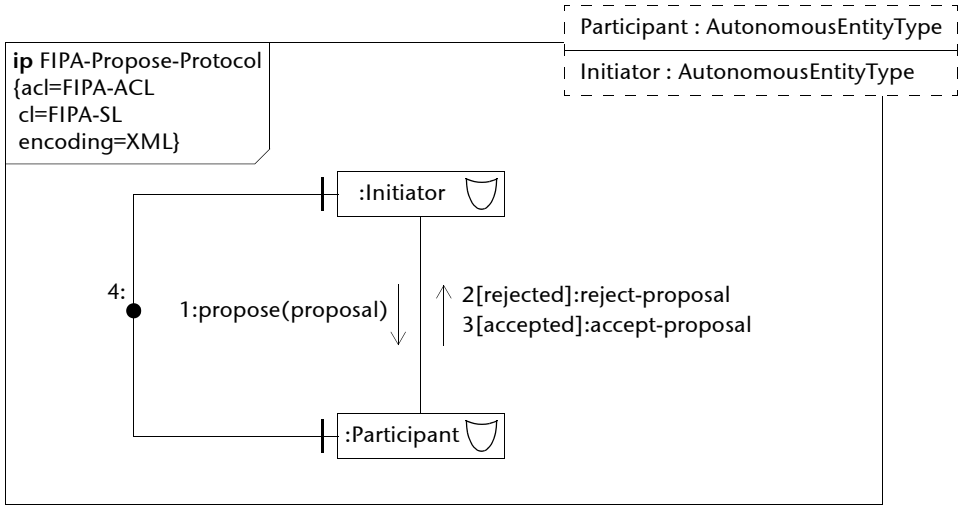


Fig. 5-72 Example of ServiceProtocol depicted as a Communication Diagram

Fig. 5-68 (p. 105) shows an example of the FIPA DirectoryFacilitator service containing several ServiceProtocols obtained by binding of the Fipa-Request-Proposal InteractionProtocol's template parameters to the actual, provider and client parameters. Semantically identical specification of the Di-

rectoryFacilitator service using the textual notation of contained ServiceProtocols is depicted in Fig. 5-73.

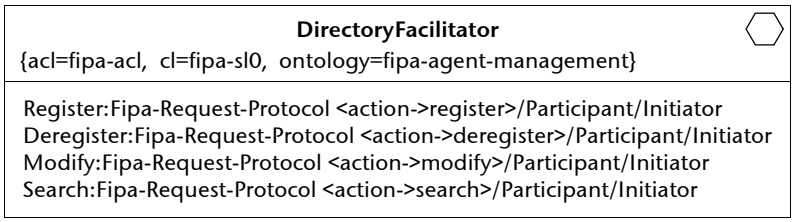


Fig. 5-73 Example of ServiceSpecification with textual notation of ServiceProtocols

Rationale ServiceProtocol is introduced to specify the parameters of an InteractionProtocol that must be bound by service providers and clients. ServiceProtocols are necessary to define ServiceSpecifications.

5.4.3 ServicedElement

Semantics ServicedElement is an abstract specialized NamedElement (from UML) used to serve as a common superclass to all the metaclasses that can provide or use services (i.e. BehavioralSemiEntitiyType, p. 60, ServicedPort, p. 110, and ServicedProperty, p. 109).

Technically, the service provision and usage is modeled by ownership of ServiceProvisions (p. 112) and ServiceUsages (p. 114).

Associations

/serviceProvision: ServiceProvision[*]	The ServiceProvision relationships owned by a ServicedElement. This is a derived association. Subsets UML Element::ownedElement and NamedElement::clientDependency.
/serviceUsage: ServiceUsage[*]	The ServiceUsage relationships owned by a ServicedElement. This is a derived association. Subsets UML Element::ownedElement and NamedElement::clientDependency.

Constraints

1. The serviceProvision meta-association refers to all client dependencies of the kind ServiceProvision:

   ```
   serviceProvision = self.clientDependency->
       select(oclIsKindOf(ServiceProvision))
   ```

2. The serviceUsage meta-association refers to all client dependencies of the kind ServiceUsage:

   ```
   serviceUsage = self.clientDependency->
       select(oclIsKindOf(ServiceUsage))
   ```

Notation There is no general notation for ServicedElement. The specific subclasses of ServicedElement define their own notation.

Rationale ServicedElement is introduced to define a common superclass for all metaclasses that may provide or require services.

5.4.4 ServicedProperty

Semantics ServicedProperty is a specialized Property (from UML) and ServicedElement (p. 108), used to model attributes that can provide or use services. It determines what services are provided and used by the behaviored semi entities when occur as attribute values of some objects.

The type of a ServicedProperty is responsible for processing or mediating incoming and outgoing communication. The ServiceProvisions (p. 112) and ServiceUsages (p. 114) owned by the the ServicedProperty are handled by its type. For details see section *5.1.1 BehavioredSemiEntityType,* p. 60.

Associations

type: BehavioredSemiEntity-Type[0..1]	The type of a ServicedProperty. Redefines UML TypedElement::type.

Notation ServicedProperty is depicted as a UML Property with the stereotype <<serviced>>.

Additionally, it can be connected to the ServiceProvision and the ServiceUsage relationships, to specify the provided and the required services respectively. See Fig. 5-74.

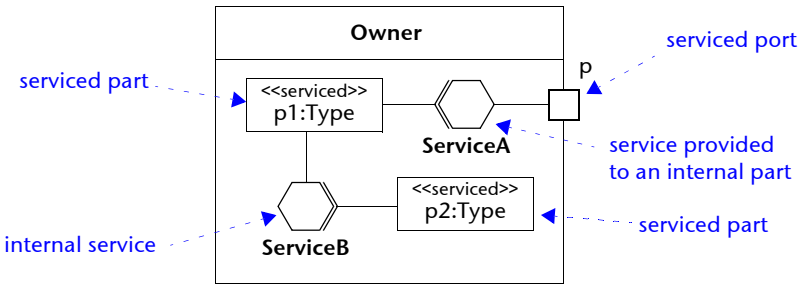


Fig. 5-74 Notation of ServicedProperty

Style guidelines The stereotype label is usually omitted and the ServicedProperty is identified only by the owned ServiceProvision and the ServiceUsage relationships.

Examples A model of a simple mobile robot is is shown in Fig. 5-75. The diagram demonstrates the use of all kinds of ServicedElements (p. 108), but does not specify details about ServiceSpecification (p. 103) bindings (for details see sections *5.4.6 ServiceProvision,* p. 112 and *5.4.7 ServiceUsage,* p. 114).

The AgentType called Robot, which models the type of a mobile robot capable of movement in a room, owns two Effectors (p. 125) (special ServicedPorts): frontWheel and rearWheels. They both provide the service Movement enabling to change position of the robot in the room, and the service WheelControl to the internal ServicedProperty called cpu, representing the

central robot controller. The frontWheel, in order to detect collisions of the robot, requires from the Room EnvironmentType a service CollisionDetection.

The Robot also contains two ServicedProperties: the already described cpu, and knowledgeBase. The knowledgeBase is used to store and manipulate a representation of the robot's environment and state. It offers a KnowledgeManipulation service that is internally used by the cpu.

The Wheel EffectorType (p. 124) is responsible for realization of the Movement and the WheelControl services, and for accessing the CollisionDetection service. Type CPU is responsible for usage of the WheelControl and the KnowledgeManipulation services. The KnowledgeBase type handles the KnowledgeManipulation service.

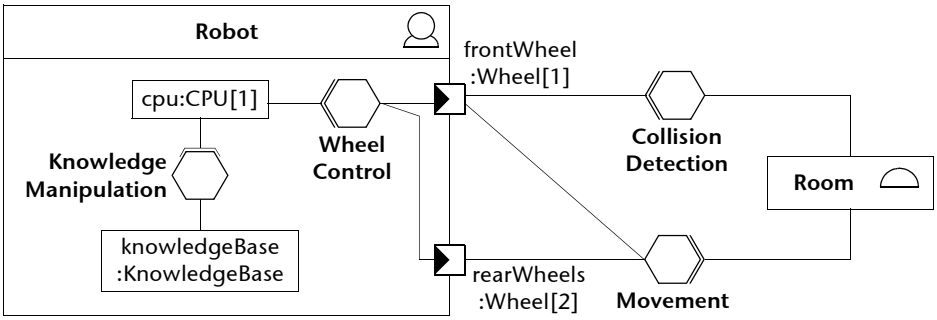


Fig. 5-75 Example of ServicedElements—simple model of a mobile robot

For another example see also Fig. 5-77 (p. 112).

Rationale ServicedProperty is introduced to model attributes that can provide or use services.

5.4.5 ServicedPort

Semantics ServicedPort is a specialized Port (from UML) and ServicedElement (p. 108) that specifies a distinct interaction point between the owning BehavioredSemiEntityType (p. 60) and other ServicedElements in the model. The nature of the interactions that may occur over a ServicedPort can, in addition to required and provided interfaces, be specified also in terms of required and provided services, particularly by associated provided and/or required ServiceSpecifications (p. 103).

The required ServiceSpecifications of a ServicedPort determine services that the owning BehavioredSemiEntityType expects from other ServicedElements and which it may access through this interaction point. The provided ServiceSpecifications determine the services that the owning BehavioredSemiEntityType offers to other ServicedElements at this interaction point.

The type of a ServicedPort is responsible for processing or mediating incoming and outgoing communication. The ServiceProvisions (p. 112) and ServiceUsages (p. 114) owned by the the ServicedPort are handled by its type. For details see section *5.1.1 BehavioredSemiEntityType* (p. 60).

Associations

type: BehavioredSemiEntity-Type[0..1]	The type of a ServicedPort. Redefines UML TypedElement::type.

Notation ServicedPort is depicted as a UML Port with the stereotype <<serviced>>.

Additionally, it can be connected to the ServiceProvision and the ServiceUsage relationships, to specify the provided and the required services respectively. See Fig. 5-76.

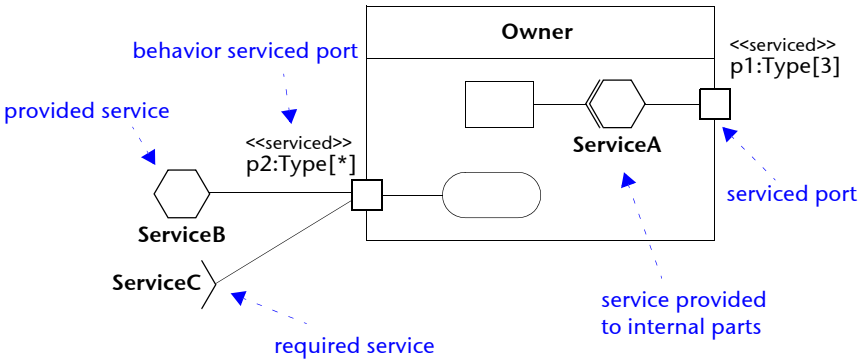


Fig. 5-76 Notation of ServicedPort

Style guidelines The stereotype label is usually omitted and the ServicedPort is identified only by the owned ServiceProvision and the ServiceUsage relationships.

Examples Fig. 5-77 shows an example of the deployment diagram of a FIPA compliant agent platform (for details see FIPA Abstract Architecture Specification at [24]). The platform is modeled as an AgentExecutionEnvironment named AgentPlatform which hosts agents. The AgentPlatform provides three services to the contained agents: DF (Directory Facilitator), AMS (Agent Management System), and MTS (Message Transport System). All these services are provided through the ServicedPort called ap4agents.

The AgentPlatform also provides services (DFProxy, AMSProxy, and MTS) to the other agent platforms over the ap2apOut (agent platform-to-agent platform output) ServicedPort, and requires the same services from other agent platforms via the ap2apIn ServicedPort. The proxy services represent restricted versions of corresponding platform-internal services.

For another example see also Fig. 5-75 (p. 110).

Rationale ServicedPort is introduced to model the distinct interaction points between the owning BehavioredSemiEntityTypes and other ServicedElements which can be used to provide and/or use services.

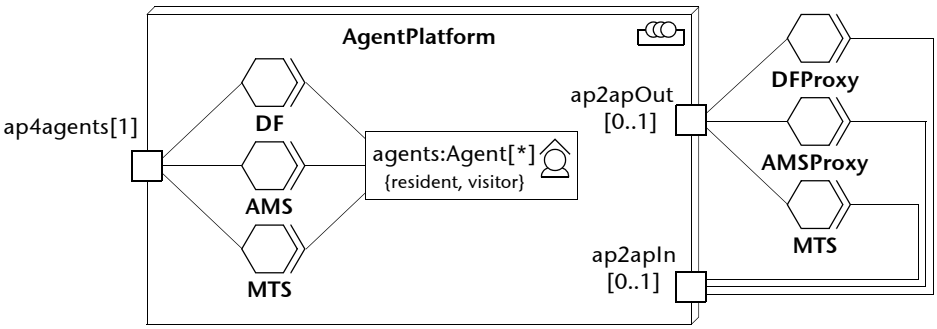


Fig. 5-77 Example of ServicedPort

5.4.6 ServiceProvision

Semantics

ServiceProvision is a specialized Realization dependency (from UML) between a ServiceSpecification (p. 103) and a ServicedElement (p. 108), used to specify that the ServicedElement provides the service specified by the related ServiceSpecification.

The details of the service provision are specified by means of owned InteractionProtocols (p. 89), which are partially bound counterparts to all ServiceProtocols (p. 105) comprised within the related ServiceSpecification.

Owned InteractionProtocols (specified by the providingIP meta-association) must bind all (and only those) template parameters of the corresponding ServiceProtocol, which are declared to be bound by a service provider.

The constraints put on bindings performed by service providers and clients of a service (see section *5.4.7 ServiceUsage,* p. 114) guarantee complementarity of those bindings. Therefore the InteractionProtocols of a ServiceProvision and a ServiceUsage, which correspond to the same ServiceSpecification, can be merged to create concrete CommunicativeInteractions (p. 89) according to which the service is accessed.

Associations

provider: ServicedElement[1]	The ServicedElement that provides the service specified by the ServiceProvision. Redefines UML Element::owner and Dependency::client.
service: ServiceSpecification[1]	The ServiceSpecification that specifies the service provided by the provider. Redefines UML Dependency::supplier.
providingIP: InteractionProtocol [1..*]	A set of InteractionProtocols each of which represents a partial binding of the related service's serviceProtocol, where all declared provider parameters are bound by the provider. Subsets UML Element::ownedElement.

Constraints

1. The providingIP binds all (and only) the provider parameters from all the service's ServiceProtocols:

   self.providingIP.templateBinding.parameterSubstitution.formal = self.service.serviceProtocol.providerParameter

Notation

ServiceProvision is depicted as a UML Dependency relationship with the stereotype <<provides>>. The ServicedElement providing the service represents a client and the ServiceSpecification is a supplier. The provider template parameter substitutions are placed in a Comment symbol attached to the relationship's arrow. See Fig. 5-78.

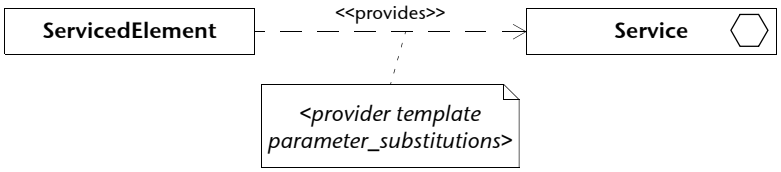


Fig. 5-78 Notation of ServiceProvision

The provider template parameter substitutions use the following syntax:

'<' *template-parameter-substitutions* '>'

The *template-parameter-substitutions* is a comma separated list of *template-parameter-substitution* defined in UML. The t*emplate-parameter-name* can be prefixed with a name of owning ServiceProtocol followed by the scope operator (::) to avoid ambiguity in naming of template parameters with the same names but from different ServiceProtocols.

Presentation options

ServiceProvision can be alternatively depicted as a solid line connecting the ServicedElement which provides the service with the ServiceSpecification, see Fig. 5-79.

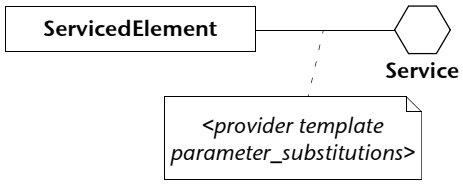


Fig. 5-79 Alternative notation of ServiceProvision

To simplify the diagram, the Comment containing the provider template parameter substitutions can be hidden.

Examples

See Fig. 5-75 (p. 110) and Fig. 5-77 (p. 112).

Rationale

ServiceProvision is introduced to specify that the ServicedElement provides the service specified by the related ServiceSpecification.

5.4.7 ServiceUsage

Semantics

ServiceUsage is a specialized Usage dependency (from UML) between a ServiceSpecification (p. 103) and a ServicedElement (p. 108), used to specify that the ServicedElement uses or requires (can request) the service specified by the related ServiceSpecification.

The details of the service usage are specified by means of owned InteractionProtocols (p. 89), which are partially bound counterparts to all ServiceProtocols (p. 105) comprised within the related ServiceSpecification.

Owned InteractionProtocols (specified by the usageIP meta-association) must bind all (and only those) template parameters of the corresponding ServiceProtocol, which are declared to be bound by a client of the service.

The constraints put on bindings performed by service providers (see section *5.4.6 ServiceProvision,* p. 112) and clients of a service guarantee complementarity of those bindings. Therefore the InteractionProtocols of a ServiceProvision and a ServiceUsage, which correspond to the same ServiceSpecification, can be merged to create concrete CommunicativeInteractions (p. 89) according to which the service is accessed.

Associations

client: ServicedElement[1]	The ServicedElement that uses the service specified by the ServiceUsage. Redefines UML Element::owner and Dependency::client.
service: ServiceSpecification[1]	The ServiceSpecification that specifies the service used by the client. Redefines UML Dependency::supplier.
usageIP: InteractionProtocol [1..*]	A set of InteractionProtocols each of which represents a partial binding of related service's serviceProtocol, where all declared client parameters are bound by the client. Subsets UML Element::ownedElement.

Constraints

1. The usageIP binds all and only the client parameters from all service's ServiceProtocols:

   self.usageIP.templateBinding.parameterSubstitution.formal =
   self.service.serviceProtocol.clientParameter

Notation

ServiceUsage is depicted as a UML Dependency relationship with the stereotype <<uses>>. The ServicedElement using the service represents a client and the ServiceSpecification is a supplier. The client template parameter

substitutions are placed in a Comment symbol attached to the relationship's arrow. See Fig. 5-80.

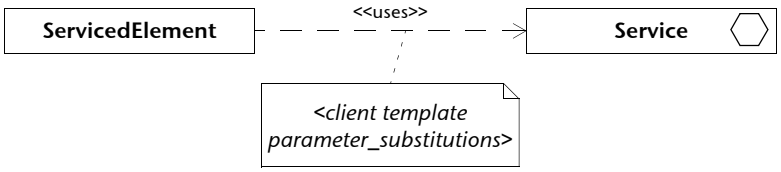


Fig. 5-80 Notation of ServiceUsage

The client template parameter substitutions use the same notation as the provider template parameter substitutions described in *5.4.6 ServiceProvision* (p. 112).

Presentation options The ServiceUsage relationship can also be shown by representing the used ServiceSpecification by an angular icon, called *service socket*, labeled with the name of the ServiceSpecification, attached by a solid line to the ServicedElement that uses this ServiceSpecification. See Fig. 5-81.

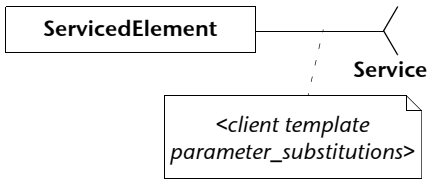


Fig. 5-81 Alternative notation of ServiceUsage

Where two ServicedElements provide and require the same ServiceSpecification, respectively, the iconic notations may be combined as shown in Fig. 5-82. This notation hints at that the ServiceSpecification in question serves to mediate interactions between the two ServicedElements.

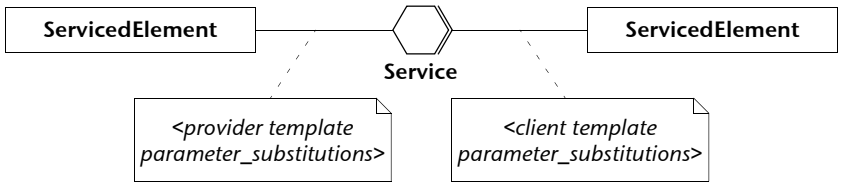


Fig. 5-82 Combination of ServiceProvision with ServiceUsage

To simplify the diagram, the Comment containing the client template parameter substitutions can be hidden.

Examples See Fig. 5-75 (p. 110) and Fig. 5-77 (p. 112).

Rationale ServiceUsage is introduced to specify that the ServicedElement uses the service specified by the related ServiceSpecification.

5.5 Observations and Effecting Interactions

Overview The *Observations and Effecting Interactions* package defines metaclasses used to model structural and behavioral aspects of observations (i.e. the ability of entities to observe features of other entities) and effecting interactions (i.e. the ability of entities to manipulate or modify the state of other entities).

Abstract syntax The diagrams of the Observations and Effecting Interactions package are shown in figures Fig. 5-83 to Fig. 5-90.

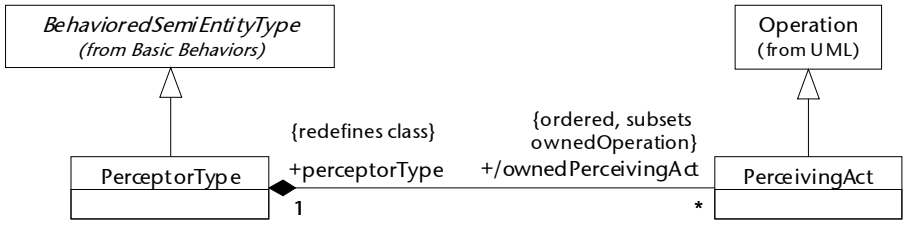


Fig. 5-83 Observations and Effecting Interactions—perceiving act and perceptor type

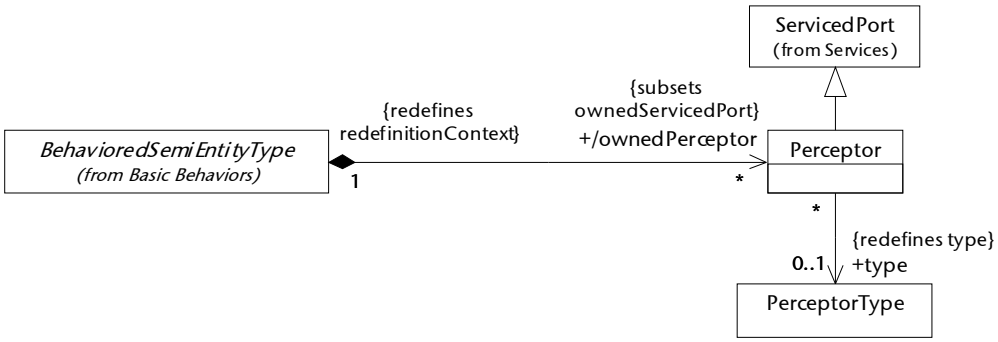


Fig. 5-84 Observations and Effecting Interactions—perceptor

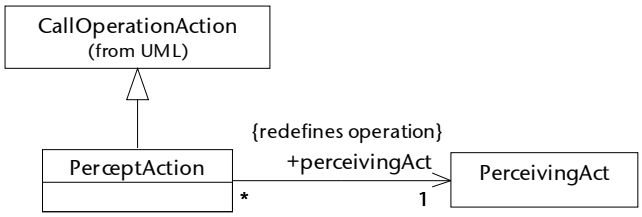


Fig. 5-85 Observations and Effecting Interactions—percept action

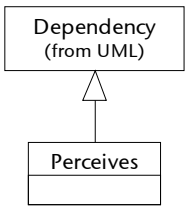


Fig. 5-86 Observations and Effecting Interactions—perceives

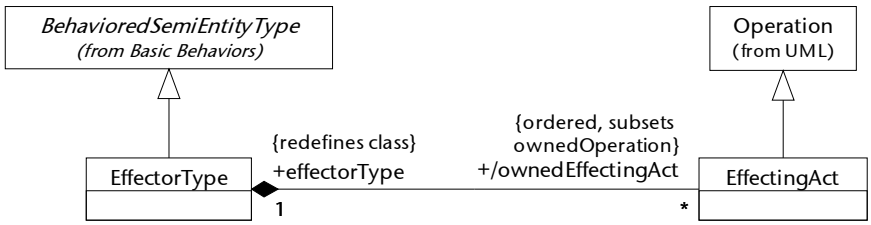


Fig. 5-87 Observations and Effecting Interactions—effecting act and effector type

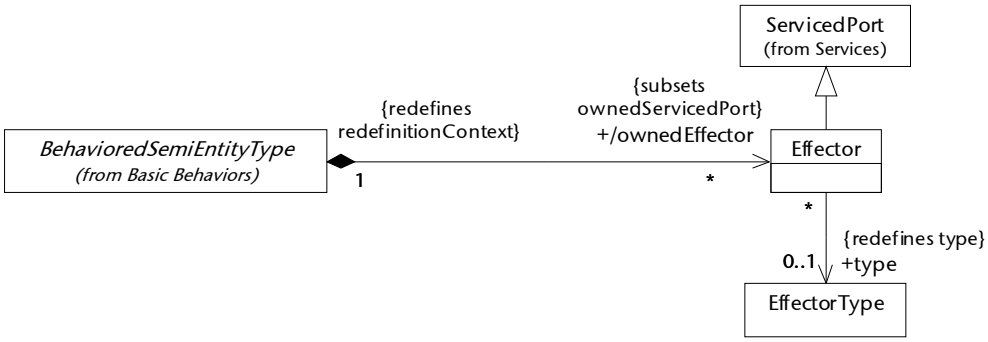


Fig. 5-88 Observations and Effecting Interactions—effector

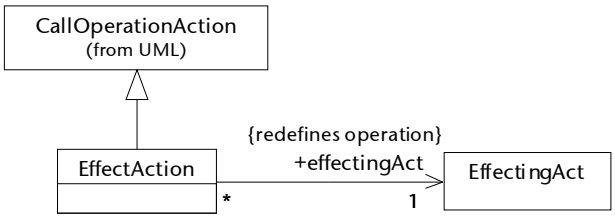


Fig. 5-89 Observations and Effecting Interactions—effect action

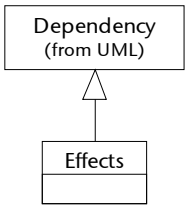


Fig. 5-90 Observations and Effecting Interactions—effects

5.5.1 PerceivingAct

Semantics PerceivingAct is a specialized Operation (from UML) which is owned by a PerceptorType (p. 118) and thus can be used to specify what perceptions the owning PerceptorType, or a Perceptor (p. 119) of that PerceptorType, can perform.

Associations

perceptorType: PerceptorType[1]	The PerceptorType that owns this PerceivingAct. Redefines UML Operation::class.

Notation PerceivingAct is shown using the same notation as for UML Operation with the stereotype <<pa>> , see Fig. 5-91.

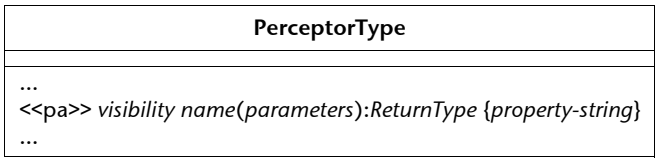


Fig. 5-91 Notation of PerceivingAct

Presentation options PerceivingAct can be placed in a special class compartment of the owning PerceptorType named <<perceiving acts>>. The stereotype <<pa>> of a particular PerceivingAct is in this case omitted. See Fig. 5-92.

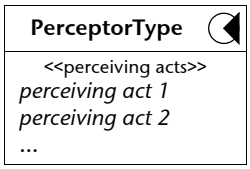


Fig. 5-92 Alternative notation of PerceivingAct—placed in a special class compartment

Examples See Fig. 5-95 (p. 119).

Rationale PerceivingAct is introduced to specify which perceptions the owning PerceptorType, or a Perceptor of that PerceptorType, can perform.

5.5.2 PerceptorType

Semantics PerceptorType is a specialized BehavioredSemiEntityType (p. 60) used to model the type of Perceptors (p. 119), in terms of owned:

- Receptions (from UML) and
- PerceivingActs (p. 117).

Associations

/ownedPerceivingAct: PerceivingAct[*]	A set of all PerceivingActs owned by the PerceptorType. The association is ordered and derived. Subsets UML Class::ownedOperation.

Constraints

1. The ownedPerceivingAct meta-association refers to all ownedOperations of the kind PerceivingAct:

   ownedPerceivingAct = self.ownedOperation->
   select(oclIsKindOf(PerceivingAct))

Notation PerceptorType is depicted as a UML Class with the stereotype label <<perceptor type>> and/or a special icon, see Fig. 5-93.

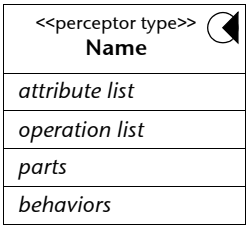


Fig. 5-93 Notation of PerceptorType

Presentation options Alternatively, the PerceptorType can, for owned PerceivingActs and Receptions, specify special compartments marked with the <<perceiving acts>> keyword, and the <<signals>> keyword respectively. Stereotypes of particular PerceivingActs and Receptions are omitted in this case. See Fig. 5-94.

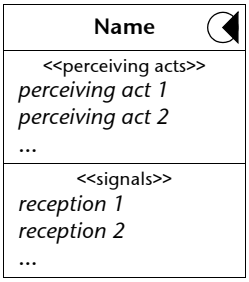


Fig. 5-94 Alternative notation of PerceptorType

Examples Fig. 5-95 shows a specification of the Eye, which is a PerceptorType used to provide information about status of the environment where the Robot, described in Fig. 5-98 (p. 121), is placed. The Eye provides two PerceivingActs (isOnFloor and isOnCube) returning an information about positions of cubes, and can process the signal newCubeAppeared, which is raised when a new cube appears in the environment.

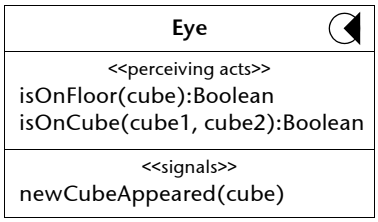


Fig. 5-95 Example of PerceptorType

Rationale PerceptorType is introduced to model types of Perceptors.

5.5.3 Perceptor

Semantics Perceptor is a specialized ServicedPort (p. 110) used to model capability of its owner (a BehavioredSemiEntityType, p. 60) to observe, i.e. perceive a state of and/or to receive a signal from observed objects. What observa-

tions a Perceptor is capable of is specified by its type, i.e. PerceptorType (p. 118).

Associations

type: PerceptorType[0..1]	The type of a Perceptor. Redefines UML TypedElement::type.

Notation Perceptor is depicted as a ServicedPort with the stereotype <<perceptor>>. See Fig. 5-96.

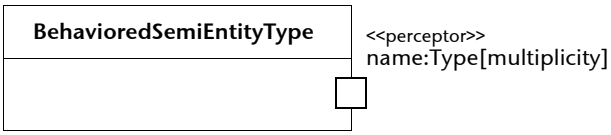


Fig. 5-96 Notation of Perceptor

Presentation options Perceptor can also be depicted as a ServicedPort with a small filled triangle pointing from the outside to the middle of owner's shape. The stereotype label is usually omitted in this case. See Fig. 5-97.

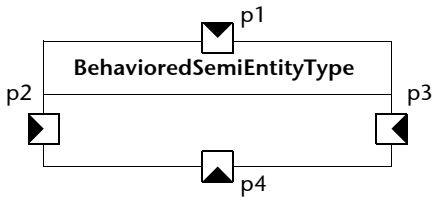


Fig. 5-97 Alternative notation of Perceptor

Examples Fig. 5-98 (a) shows the illustration of a robot able to move (to change position and direction), see its environment and manipulate the cubes placed in the environment. The robot for this purpose is equipped with three wheels, two "eyes" (cameras with a picture recognition system able to perceive and reason about the environment), and one manipulator used to move and lift the cubes.

Fig. 5-98 (b) shows a model of the robot. It is modeled as an AgentType with attributes position and direction, and respective Perceptors (eyes) and Effectors (wheels and manipulator). The diagram also depicts Perceives and Effects relationships between the robot and cubes (meaning that the robot is able to see and to manipulate the cubes), and the robot itself (because it is able to change its own position and direction by means of wheels).

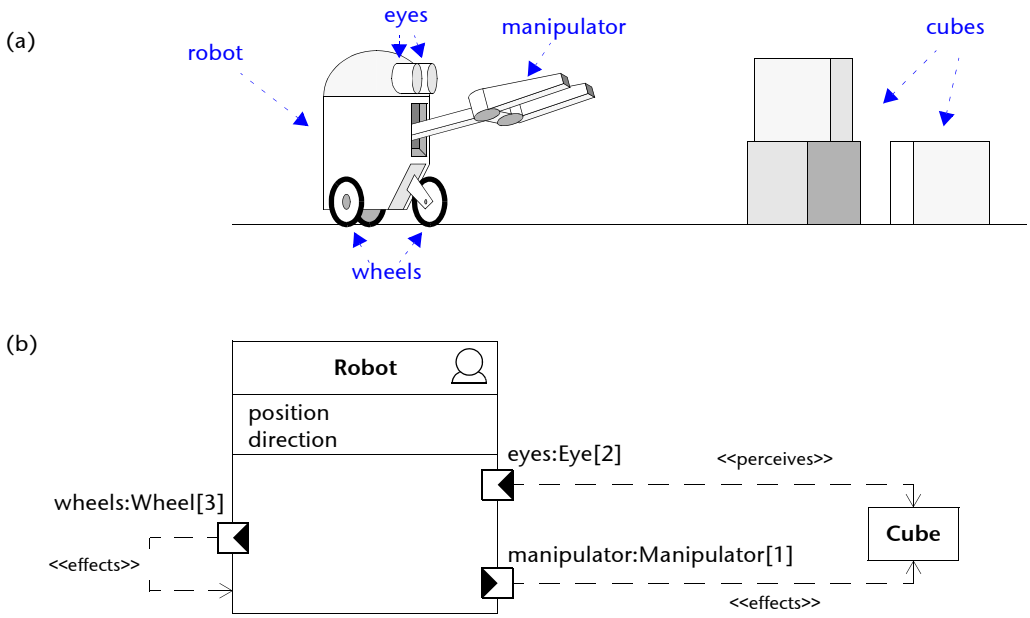


Fig. 5-98 Example of non-communicative interaction modeling—mobile robot: *(a) illustration of the robot, (b) AML model of the robot.*

For more examples see also Fig. 4-5 (p. 26), Fig. 4-32 (p. 41), and Fig. 5-7 (p. 66).

Rationale Perceptor is introduced to model the capability of its owner to observe.

5.5.4 PerceptAction

Semantics PerceptAction is a specialized CallOperationAction (from UML) which can call PerceivingActs (p. 117). As such, PerceptAction can transmit an operation call request to a PerceivingAct, what causes the invocation of the associated behavior.

PerceptAction being a CallOperationAction allows to call PerceivingActs both synchronously and asynchronously.

Associations

perceivingAct: PerceivingAct[1]	The PerceivingAct to be invoked by the action execution. Redefines UML CallOperationAction::operation.

Notation PerceptAction is drawn as a UML CallOperationAction with the stereotype <<percept>> and/or a special icon, see Fig. 5-99. Action's name represents a PerceivingAct's call, i.e. it's name and possibly also parameter values.

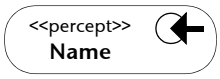


Fig. 5-99 Notation of PerceptAction

Presentation options If the action's name is different from the PerceivingAct's call, the call may appear in parentheses below the action's name. To indicate the PerceptorType (p. 118) owning the called PerceivingAct, it's name postfixed by a double colon may precede the PerceivingAct's call. See Fig. 5-52.

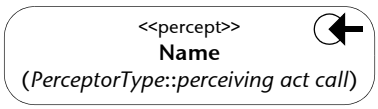


Fig. 5-100 Alternative notation of PerceptAction—indication of PerceivingAct's call

PerceptAction can use all presentation options from UML CallOperationAction.

Examples Fig. 5-101 shows an example of the Plan (p. 156) describing how the robot, as in Fig. 5-98 (p. 121), can achieve the placement of the red cube on the blue one. The application logic is described solely by the PerceptActions (isOnFloor) and the EffectActions (putOn and putOnFloor).

For details on how the called PerceivingAct and EffectingActs are defined see Fig. 5-95 (p. 119) and Fig. 5-107 (p. 125) respectively.

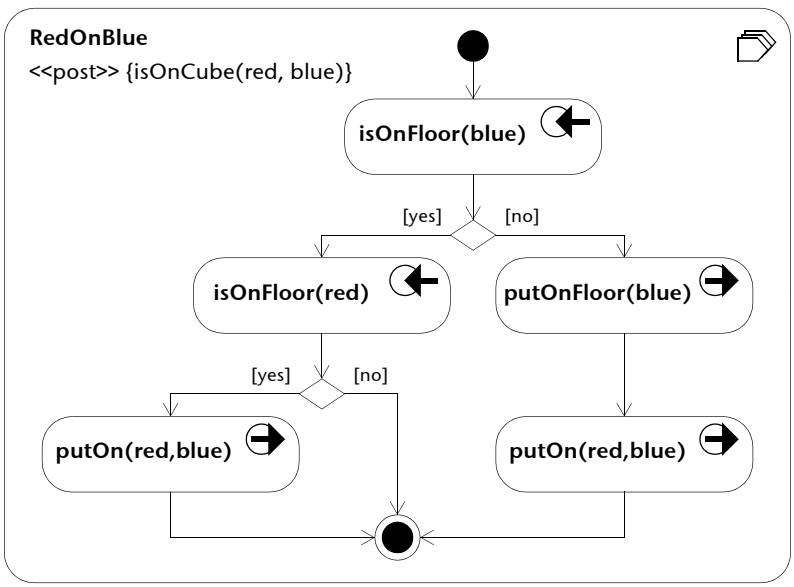


Fig. 5-101 Example of Plan, PerceptAction and EffectAction

Rationale PerceptAction is introduced to model observations in Activities.

5.5.5 Perceives

Semantics Perceives is a specialized Dependency (from UML) used to model which elements can observe others.

Suppliers of the Perceives dependency are the observed elements, particularly NamedElements (from UML).

Clients of the Perceives dependency represent the objects that observe. They are usually modeled as:

- ❒ BehavioredSemiEntityTypes (p. 60),
- ❒ PerceivingActs (p. 117),
- ❒ PerceptorTypes (p. 118), or
- ❒ Perceptors (p. 119).

Notation Perceives is depicted as a UML Dependency relationship with the stereotype <<perceives>>. See Fig. 5-102.

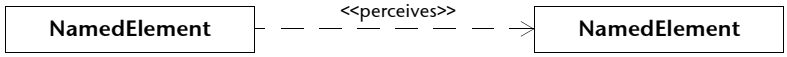


Fig. 5-102 Notation of Perceives

Presentation options Alternatively, the Perceives dependency can be depicted as a dashed line with a filled triangle as an arrowhead. One side of the triangle is oriented toward the icon of the observed element. The stereotype label is usually omitted in this case. See Fig. 5-103.

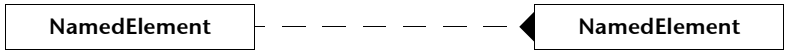


Fig. 5-103 Alternative notation of Perceives

Examples See Fig. 5-98 (p. 121).

Rationale Perceives is introduced to model which elements can observe others.

5.5.6 EffectingAct

Semantics EffectingAct is a specialized Operation (from UML) which is owned by an EffectorType (p. 124) and thus can be used to specify what effecting acts the owning EffectorType, or an Effector (p. 125) of that EffectorType, can perform.

Associations

effectorType: EffectorType[1]	The EffectorType that owns this EffectingAct. Redefines UML Operation::class.

Notation EffectingAct is shown using the same notation as for UML Operation with the stereotype <<ea>> , see Fig. 5-104.

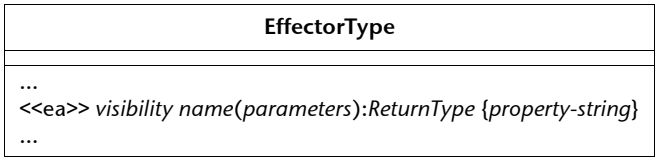


Fig. 5-104 Notation of EffectingAct

Presentation options EffectingAct can be placed in a special class compartment of the owning EffectorType named <<effecting acts>>. The stereotype <<ea>> of a particular EffectingAct is in this case omitted. See Fig. 5-105.

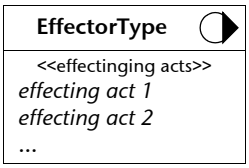


Fig. 5-105 Alternative notation of EffectingAct—placed in a special class compartment

Examples See Fig. 5-107 (p. 125).

Rationale EffectingAct is introduced to specify which effecting acts the owning EffectorType, or an Effector of that EffectorType, can perform.

5.5.7 EffectorType

Semantics EffectorType is a specialized BehavioredSemiEntityType (p. 60) used to model type of Effectors (p. 125), in terms of owned EffectingActs (p. 123).

Associations

/ownedEffectingAct: EffectingAct[*]	A set of all EffectingActs owned by the EffectorType. The association is ordered and derived. Subsets UML Class::ownedOperation.

Constraints

1. The ownedEffectingAct meta-association refers to all ownedOperations of the kind EffectingAct:

   ownedEffectingAct = self.ownedOperation->
   select(oclIsKindOf(EffectingAct))

Notation EffectorType is depicted as a UML Class with the stereotype label <<effector type>> and/or a special icon, see Fig. 5-106.

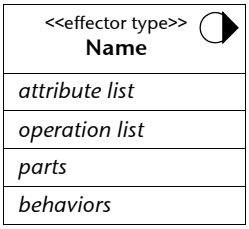


Fig. 5-106 Notation of EffectorType

Presentation options Alternatively, the EffectorType can for owned EffectingActs specify a special compartment marked with the <<effecting acts>> keyword. See Fig. 5-105.

Examples Fig. 5-107 shows a specification of the Manipulator, which is an EffectorType used to manipulate the cubes placed in the environment of the Robot, described in Fig. 5-98 (p. 121). The Manipulator provides two EffectingActs: putOnFloor enabling the placement of a cube on the floor, and putOn enabling the placement of one cube on another.

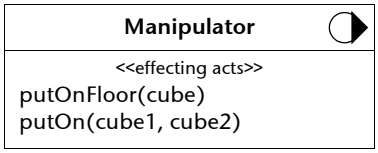


Fig. 5-107 Example of EffectorType

Rationale EffectorType is introduced to model types of Effectors.

5.5.8 Effector

Semantics Effector is a specialized ServicedPort (p. 110) used to model the capability of its owner (a BehavioredSemiEntityType, p. 60) to bring about an effect on others, i.e. to directly manipulate with (or modify a state of) some other objects. What effects an Effector is capable of is specified by its type, i.e. EffectorType (p. 124).

Associations

type: EffectorType[0..1]	The type of an Effector. Redefines UML TypedElement::type.

Notation Effector is depicted as a ServicedPort with the stereotype <<effector>>. See Fig. 5-108.

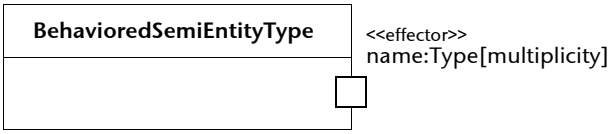


Fig. 5-108 Notation of Effector

Presentation options Effector can also be depicted as a ServicedPort with a small filled triangle pointing from middle of the owner's shape to the outside. The stereotype label is usually omitted in this case. See Fig. 5-97.

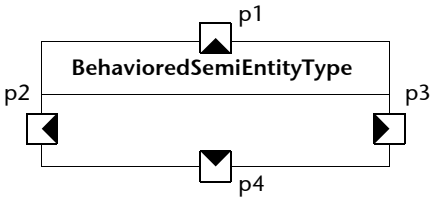


Fig. 5-109 Alternative notation of Effector

Examples See Fig. 4-5 (p. 26), Fig. 4-32 (p. 41), Fig. 5-75 (p. 110), and Fig. 5-98 (p. 121).

Rationale Effector is introduced to model the capability of its owner to bring about an effect on other objects.

5.5.9 EffectAction

Semantics EffectAction is a specialized CallOperationAction (from UML) which can call EffectingActs (p. 123). Thus, an EffectAction can transmit an operation call request to an EffectingAct, which causes the invocation of the associated behavior.

EffectAction being a CallOperationAction allows calling EffectingActs both synchronously and asynchronously.

Associations

effectingAct: EffectingAct[1]	The EffectingAct to be invoked by the action execution. Redefines UML CallOperationAction::operation.

Notation EffectAction is drawn as a UML CallOperationAction with the stereotype <<effect>> and/or a special icon, see Fig. 5-110. The action's name represents an EffectingAct's call, i.e. it's name and possibly also parameter values.

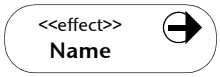


Fig. 5-110 Notation of EffectAction

Presentation options If the action's name is different from the EffectingAct's call, the call may appear in parentheses below the action's name. To indicate the EffectorType

(p. 124) owning the called EffectingAct, it's name postfixed by a double colon may precede the EffectingAct's call. See Fig. 5-111.

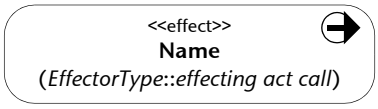


Fig. 5-111 Alternative notation of EffectAction—indication of EffectingAct's call

EffectAction can use all presentation options from UML CallOperationAction.

Examples See Fig. 5-101 (p. 122).

Rationale EffectAction is introduced to model effections in Activities.

5.5.10 Effects

Semantics Effects is a specialized Dependency (from UML) used to model which elements can effect others.

Suppliers of the Effects dependency are the effected elements, particularly NamedElements (from UML).

Clients of the Effects dependency represent the objects which effect. They are usually modeled as:

- BehavioredSemiEntityTypes (p. 60),
- EffectingActs (p. 123),
- EffectorTypes (p. 124), or
- Effectors (p. 125).

Notation Effects is depicted as a UML Dependency relationship with the stereotype <<effects>>. See Fig. 5-112.

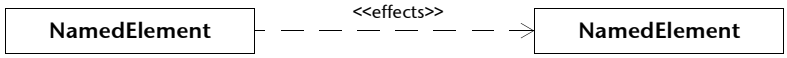


Fig. 5-112 Notation of Effects

Presentation options Alternatively, the Effects dependency can be depicted as a dashed line with a filled triangle as an arrowhead. One point of the triangle touches an icon of the the observed element. The stereotype label is usually omitted in this case. See Fig. 5-113.

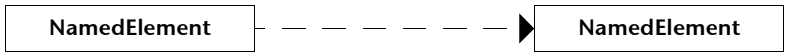


Fig. 5-113 Alternative notation of Effects

Examples See Fig. 5-98 (p. 121).

Rationale Effects is introduced to model which elements can effect others.

5.6 Mobility

Overview The *Mobility* package defines metaclasses used to model structural and behavioral aspects of entity mobility[7].

Abstract syntax The diagrams of the Mobility package are shown in figures Fig. 5-114 and Fig. 5-115.

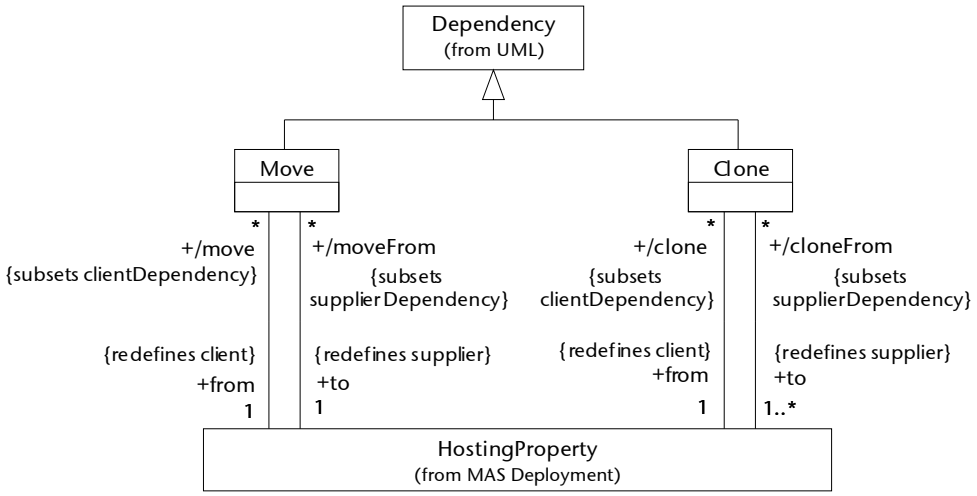


Fig. 5-114 Mobility—mobility relationships

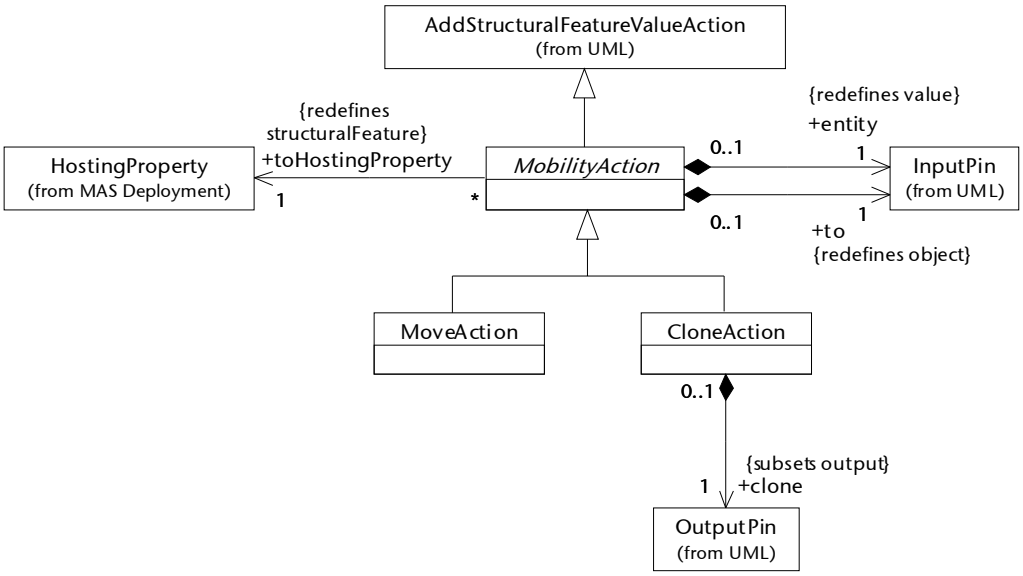


Fig. 5-115 Mobility—mobility-related actions

[7] Agents are special entities and therefore agent mobility is also covered by the AML mobility mechanisms.

5.6.1 Move

Semantics Move is a specialized Dependency (from UML) between two HostingProperties (p. 51) used to specify that the entities represented by the source HostingProperty (specified by the from meta-association) can be moved/transferred to the instances of the AgentExecutionEnvironment (p. 49) owning the destination HostingProperty (specified by the to meta-association).

For example, a Move dependency between a HostingProperty A of type T (owned by AgentExecutionEnvironment AEE1) and a HostingProperty B of the same type T (owned by another AgentExecutionEnvironment AEE2) means that entities of type T can be moved from AEE1 to AEE2.

AML does not specify the type and other details of moving, which may be technology dependent. If needed, they can be specified as specific tagged values, comments, constraints, linked/attached information, etc.

Associations

from: HostingProperty[1]	The HostingProperty representing the source of moving. Redefines UML Dependency::client.
to: HostingProperty[1]	The HostingProperty representing the destination of moving. Redefines UML Dependency::supplier.

Constraints

1. If both specified, the type of the HostingProperty referred to by the to meta-association must conform to the type of the HostingProperty referred to by the from meta-association:

   (self.from.type->notEmpty() and self.to.type->notEmpty()) implies
   self.to.type.conformsTo(self.from.type)

Notation Move is depicted as a UML Dependency relationship with the stereotype <<move>>. The from HostingProperty is specified as a dependency client and the to HostingProperty is a dependency supplier. See Fig. 5-116.

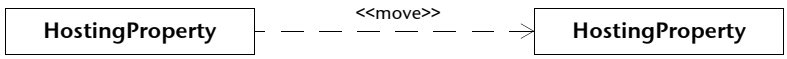


Fig. 5-116 Notation of Move

Examples Fig. 5-117 extends the example from Fig. 4-47 (p. 50) by modeling load balancing and agent mobility. Agents of type Broker residing in the TradingClient can be cloned to the TradingServer instances running at the MainStockExchangeServers. Each TradingServer can control the load of its machine, and if necessary, it can move some brokers (agents of type Broker) to any of the BackupStockExchangeServers. The LoadBalanceManager agents are concerned with monitoring of machine load and moving brokers to other machines. The brokers can move back to the MainStockExchangeServers.

Rationale Move is introduced to model the movement of entities between instances of AgentExecutionEnvironments.

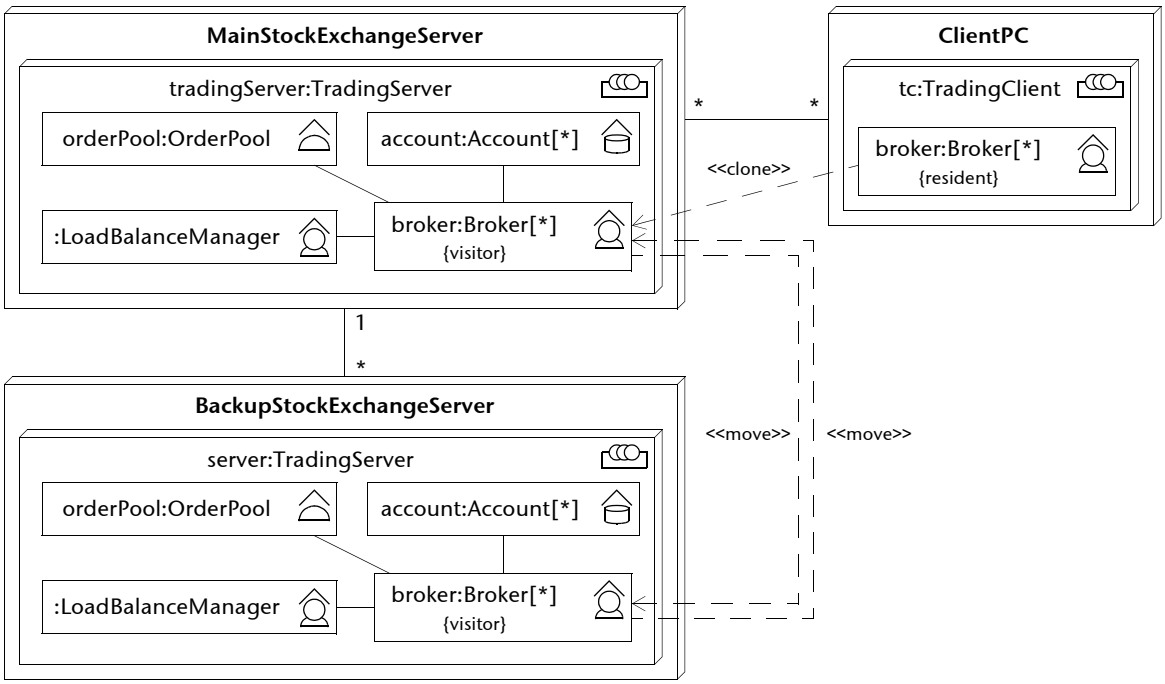


Fig. 5-117 Example of Move and Clone relationships

5.6.2 Clone

Semantics

Clone is a specialized Dependency (from UML) between HostingProperties (p. 51) used to specify that entities represented by the source HostingProperty (specified by the from meta-association) can be cloned to the instances of the AgentExecutionEnvironment (p. 49) owning the destination HostingProperties (specified by the to meta-association).

For example, a Clone dependency between a HostingProperty A of type T (owned by AgentExecutionEnvironment AEE1) and a HostingProperty B of the same type T (owned by another AgentExecutionEnvironment AEE2) means that entities of type T can be cloned from AEE1 to AEE2.

AML does not specify the type and other details of cloning, which may be technology dependent. If needed, they can be specified as specific tagged values, comments, constraints, linked/attached information, etc.

Associations

from: HostingProperty[1]	The HostingProperty representing the source of cloning. Redefines UML Dependency::client.
to: HostingProperty[1..*]	The HostingProperties representing the destination of cloning. Redefines UML Dependency::supplier.

Constraints

1. If specified, the types of the HostingProperties referred to by the to meta-association must conform to the type of the HostingProperty referred to by the from meta-association:

```
(self.from.type->notEmpty() and self.to.type->notEmpty()) implies
self.to.type->forAll(conformsTo(self.from.type))
```

Notation Clone is depicted as a UML Dependency relationship with the stereotype <<clone>>. The from HostingProperty is specified as a dependency client and the to HostingProperty is a dependency supplier. See Fig. 5-118.

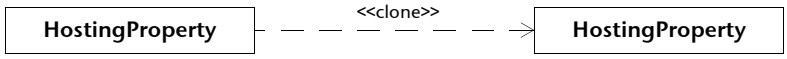


Fig. 5-118 Notation of Clone

Examples See Fig. 5-117 (p. 130).

Rationale Clone is introduced to model the cloning of entities among instances of AgentExecutionEnvironments.

5.6.3 MobilityAction

Semantics MobilityAction is an abstract specialized AddStructuralFeatureValueAction (from UML) used to model mobility actions of entities, i.e. actions that cause movement or cloning of an entity from one AgentExecutionEnvironment (p. 49) to another one. MobilityAction specifies:

- which entity is being moved or cloned (entity meta-association),
- the destination AgentExecutionEnvironment instance where the entity is being moved or cloned (to meta-association), and
- the HostingProperty (p. 51) owned by the destination AgentExecutionEnvironment, where the moved or cloned entity is being placed (toHostingProperty meta-association).

If the destination HostingProperty is ordered, the insertAt meta-association (inherited from AddStructuralFeatureValueAction) specifies the position at which to insert the entity.

MobilityAction has two concrete subclasses:

- MoveAction (p. 132) and
- CloneAction (p. 133).

Associations

entity: InputPin[1]	The InputPin specifying the entity being moved or cloned. Redefines UML WriteStructuralFeatureValueAction::value.
to: InputPin[1]	The InputPin specifying the destination AgentExecutionEnvironment instance where the entity is being moved or cloned. Redefines UML StructuralFeatureAction::object.

toHostingProperty: HostingProperty[1]	The HostingProperty where the moved or cloned entity is being placed. Redefines UML StructuralFeatureAction::structuralFeature.

Constraints

1. If the type of the InputPin referred to by the entity meta-association is specified, it must be an EntityType:

   self.entity.type->notEmpty() implies
   self.entity.type.oclIsKindOf(EntityType)

2. If the type of the InputPin referred to by the to meta-association is specified, it must be an AgentExecutionEnvironment:

   self.to.type->notEmpty() implies
   self.to.type.oclIsKindOf(AgentExecutionEnvironment)

3. If the type of the InputPin referred to by the to meta-association is specified, the HostingProperty referred to by the toHostingProperty meta-association must be an owned attribute of that type:

   self.to.type->notEmpty() implies
   self.to.type.ownedAttribute->includes(self.toHostingProperty)

Notation There is no general notation for MobilityAction. The specific subclasses of MobilityAction define their own notation.

Rationale MobilityAction is introduced to define the common features of all its subclasses.

5.6.4 MoveAction

Semantics MoveAction is a specialized MobilityAction (p. 131) used to model an action that results in a removal of the entity (specified by the entity meta-association, inherited from MobilityAction) from its current hosting location, and its insertion as a value to the destination HostingProperty (specified by the toHostingProperty meta-association, inherited from MobilityAction) of the owning AgentExecutionEnvironment (p. 49) instance (specified as the to meta-association, inherited from MobilityAction).

Notation MoveAction is drawn as a UML Action with the stereotype <<move>> and/or a special icon, see Fig. 5-119.

Fig. 5-119 Notation of MoveAction

Optionally, the name of the moved entity, followed by a greater-than character ('>') and a specification of the destination may appear in parentheses below the action's name. The movement destination is specified as the name of the destination AgentExecutionEnvironment instance (given by the to meta-association), delimited by a period from the name of the destina-

tion HostingProperty (given by the toHostingProperty meta-association). If the destination HostingProperty is ordered, the value of the insertAt meta-association can be placed after the destination HostingProperty's name in brackets.

If the entity being moved is an instance which executes the MoveAction, it can be identified by the keyword 'self' in place of the entity name.

Examples The diagram in Fig. 5-120 shows the Plan (p. 156) of the AgentType (p. 25) Broker (defined in Fig. 5-117, p. 130) describing how to behave if the LoadBalanceManager (defined also in Fig. 5-117) sends a move command.

After accepting the move command, the Broker evaluates the move. If not possible, the Broker sends a reject CommunicationMessage to the LoadBalanceManager. If the movement is possible, the Broker sends an accept CommunicationMessage to the LoadBalanceManager and moves to the destination AgentExecutionEnvironment instance, specified by the moveCommand, where it is placed in the HostingProperty named broker.

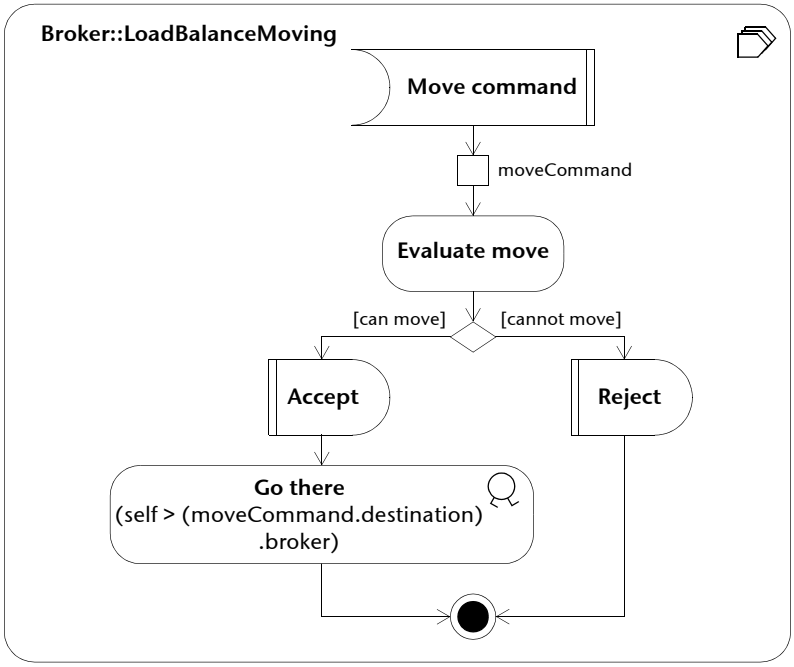


Fig. 5-120 Example of Plan and MoveAction

Rationale MoveAction is introduced to model the movement of entities in Activities.

5.6.5 CloneAction

Semantics CloneAction is a specialized MobilityAction (p. 131) used to model an action that results in a insertion of a clone of the entity (specified by the entity meta-association, inherited from MobilityAction) as a value to the destination HostingProperty (specified by the toHostingProperty meta-association, inherited from MobilityAction) of the owning AgentExecutionEnvironment (p. 49) instance (specified as the to meta-association, inherited from Mobil-

ityAction). The original entity remains running at its current hosting location.

The entity clone is represented by the action's OutputPin (specified by the clone meta-association).

Associations

clone: OutputPin[1]	The OutputPin representing the entity clone. Subsets UML Action::output.

Constraints

1. If the type of the OutputPin referred to by the clone meta-association is specified, it must be an EntityType:

   self.clone.type->notEmpty() implies
   self.clone.type.oclIsKindOf(EntityType)

2. The type of the OutputPin referred to by the clone meta-association must conform to the type of the InputPin referred to by the entity meta-association, if the both specified:

   (self.clone.type->notEmpty() and self.entity.type->notEmpty()) implies
   self.clone.type.conformsTo(self.entity.type)

Notation

CloneAction is drawn as a UML Action with the stereotype <<clone>> and/or a special icon see Fig. 5-121.

Fig. 5-121 Notation of CloneAction

Optionally, the name of the cloned entity, followed by a greater-than character ('>') and a specification of the destination may appear in parentheses below the action's name. The movement destination is specified as the name of the destination AgentExecutionEnvironment instance (given by the to meta-association), delimited by a period from the name of the destination HostingProperty (given by the toHostingProperty meta-association). If the destination HostingProperty is ordered, the value of the insertAt meta-association can be placed after the destination HostingProperty's name in brackets. The clone entity is specified as an OutputPin.

If the entity being cloned is an instance which executes the CloneAction, it can be identified by the keyword 'self' in place of the entity name.

Examples

The activity diagram in Fig. 5-122 shows the behavior of the broker agent (its type is defined in Fig. 5-117, p. 130), running at the client, after reception of a new order from a user. The broker checks the order, and if incorrect, a report is produced. In the case of a correct order, the broker selects the most appropriate trading server and clones itself to the selected server. In addition to the broker's code and status information, the clone (called brokerClone), also carries the order information.

When the brokerClone starts to run at the server, it tries to register. If unsuccessful, it announces to the broker the registration refusal which in turn reports this fact to the user. If registration was successful, the brokerClone places the order onto the orderPool (see Fig. 5-117) and negotiates the order with other brokers. If finished, the negotiation result is sent back to the broker which reports the result to the user. After sending the result, the brokerClone terminates its execution at the server.

The diagram is partitioned by two-dimensional ActivityPartitions. The first dimension contains ActivityPartitions representing entities responsible for performing actions. The second dimension contains ActivityPartitions representing the AgentExecutionEnvironment instances where the entities perform relevant actions. If the diagram was specified at the class level, the ActivityPartitions would represent EntityTypes and AgentExecutionEnvironments (and possibly also their HostingProperties) respectively. This is a usual way of partitioning mobility-related Activities, that enables the specification of which entities perform what actions at what hosting places, and also allows modeling the synchronization of activities performed by different entities.

Orientation of partitioning can be arbitrary, i.e. ActivityPartitions representing entities can be either horizontal or vertical, and the ActivityPartitions representing the hosting places should be orthogonal.

Rationale CloneAction is introduced to model the cloning of entities in Activities.

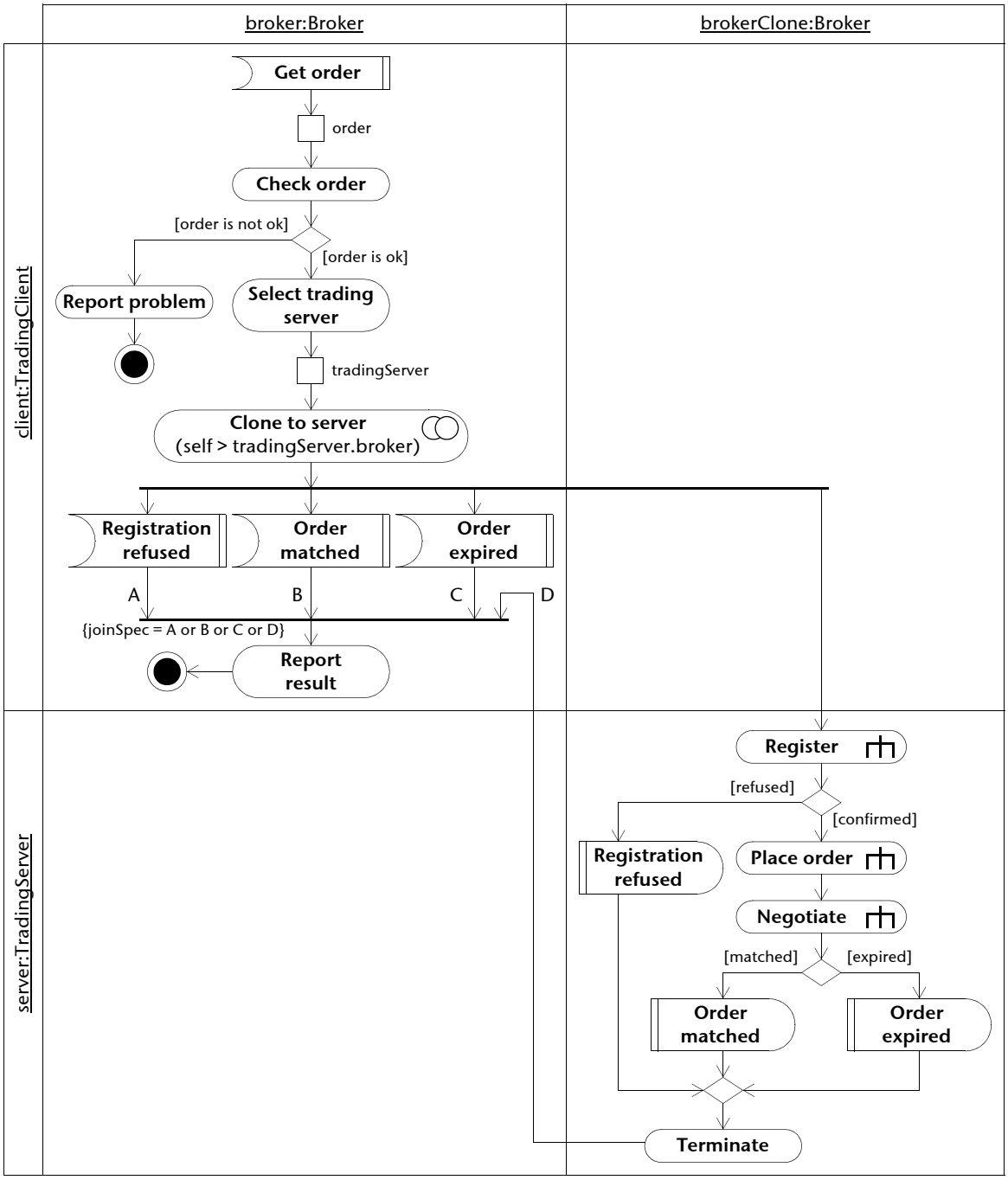


Fig. 5-122 Example of CloneAction

6 Mental

Overview The *Mental* package defines the metaclasses which can be used to:

- ❒ support analysis of complex problems/systems, particularly by:
  - modeling intentionality in use case models,
  - goal-based requirements modeling,
  - problem decomposition, etc.
- ❒ model mental attitudes of autonomous entities, which represent their informational, motivational and deliberative states.

Package structure The package diagram of the Mental package is depicted in Fig. 6-1.

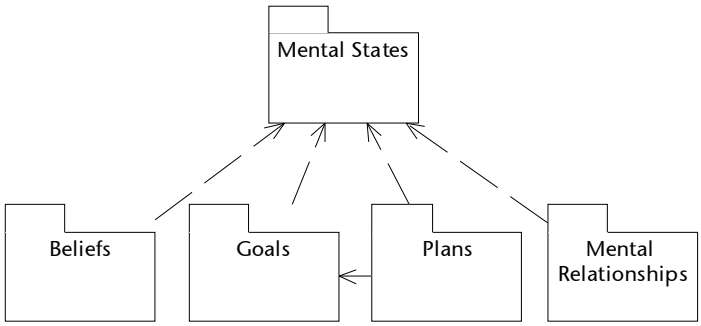


Fig. 6-1 Mental—package structure

6.1 Mental States

Overview The *Mental States* package defines the core metaclasses used to define concrete metaclasses contained within the rest of the Mental sub-packages, i.e. *6.2 Beliefs* (p. 149), *6.3 Goals* (p. 151), *6.4 Plans* (p. 155) and *6.5 Mental Relationships* (p. 162).

Abstract syntax The diagrams of the Mental State package are shown in figures Fig. 6-2 to Fig. 6-5.

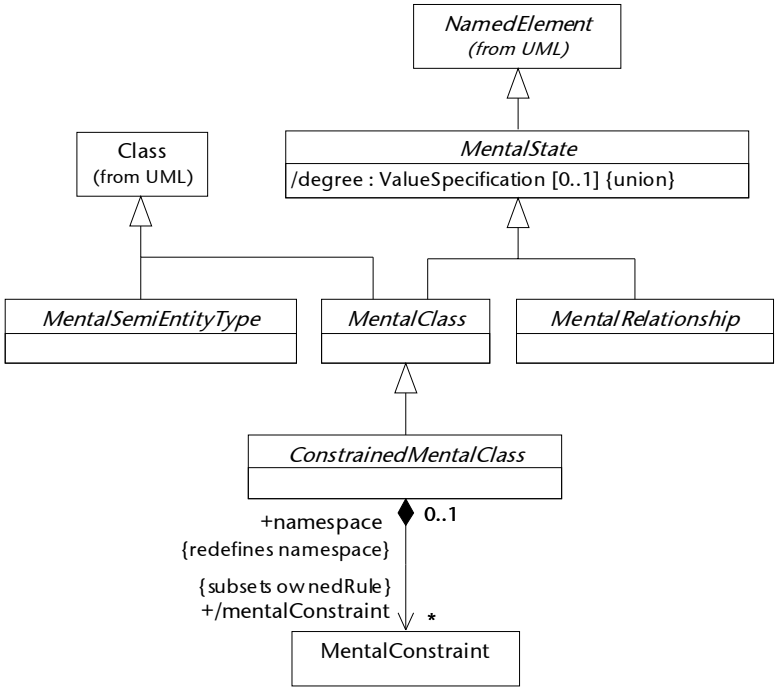


Fig. 6-2 Mental States—mental states and mental semi-entity type

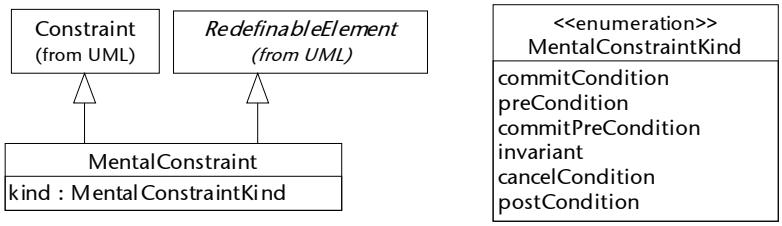


Fig. 6-3 Mental States—mental constraint

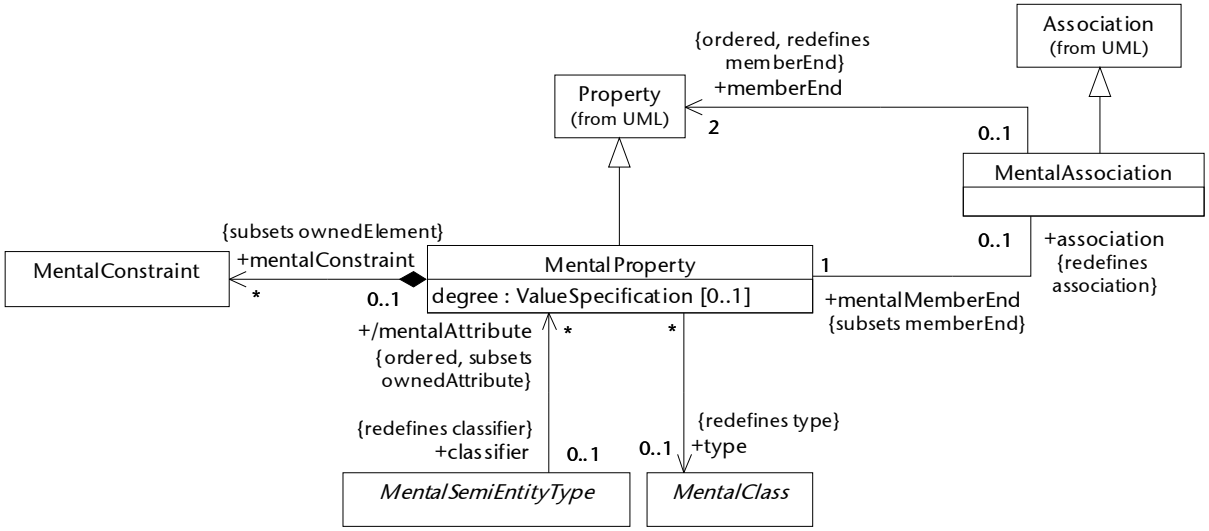


Fig. 6-4 Mental States—mental property and mental association

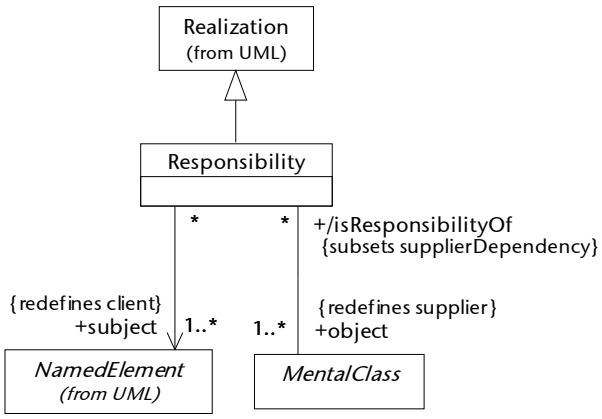


Fig. 6-5 Mental States—responsibility

6.1.1 MentalState

Semantics MentalState is an abstract specialized NamedElement (from UML) serving as a common superclass to all metaclasses which can be used for:

- ❒ modeling mental attitudes of MentalSemiEntityTypes (p. 143), which represent their informational, motivational and deliberative states, and
- ❒ support for the human mental process of requirements specification and analysis of complex problems/systems, particularly by:
  - expressing intentionality in use case models,
  - goal-based requirements modeling,
  - problem decomposition, etc.

Attributes

/degree: ValueSpecification [0..1]	The degree of a MentalState. Its specific semantics varies depending on the context of MentalState's subclasses that subset it. This is a derived union.

Notation There is no general notation for MentalState. The specific subclasses of MentalState define their own notation.

Rationale MentalState is introduced to define the common features of all its subclasses.

6.1.2 MentalClass

Semantics MentalClass is an abstract specialized Class (from UML) and MentalState (p. 139) serving as a common superclass to all the metaclasses which can be used to specify mental attitudes of MentalSemiEntityTypes (p. 143). Technically, MentalProperties (p. 144) can only be of the MentalClass type.

Furthermore, the object meta-association of the Responsibility relationship (p. 148) can also only be of the MentalClass type.

Associations

/isResponsibilityOf: Responsibility[*]	The Responsibility relationships that refer to the MentalClass as an object of responsibility. This is a derived association.

Constraints

1. The isResponsibilityOf meta-association refers to all supplierDependencies of the kind Responsibility:

   ```
   isResponsibilityOf = self.supplierDependency->
       select(oclIsKindOf(Responsibility))
   ```

Notation There is no general notation for MentalClass. The specific subclasses of MentalClass define their own notation.

Rationale MentalClass is introduced to specify the mental attitudes of MentalSemiEntityTypes and objects of Responsibility relationship.

6.1.3 ConstrainedMentalClass

Semantics ConstrainedMentalClass is an abstract specialized MentalClass (p. 140) which allows its concrete subclasses to specify MentalConstraints (p. 141).

Note: To avoid misinterpretation of a set of multiple MentalConstraints of the same kind defined within one ConstrainedMentalClass, AML allows the specification of only one MentalConstraint of a given kind within one ConstrainedMentalClass.

Associations

/mentalConstraint: MentalConstraint[*]	A set of the MentalConstraints owned by the ConstrainedMentalClass. This is a derived association. Subsets UML Namespace::ownedRule.

Constraints

1. Each mentalConstraint must have a different kind:

   self.mentalConstraint->forAll(mc1, mc2 | mc1.kind<>mc2.kind)

2. The mentalConstraint meta-association refers to all ownedRules of the kind MentalConstraint:

   mentalConstraint = self.ownedRule->
   select(oclIsKindOf(MentalConstraint))

Notation

There is no general notation for ConstrainedMentalClass. The specific subclasses of ConstrainedMentalClass define their own notation.

Rationale

ConstrainedMentalClass is introduced to allow the specification of MentalConstraints for all its subclasses.

6.1.4 MentalConstraint

Semantics

MentalConstraint is a specialized Constraint (from UML) and RedefinableElement (from UML), used to specify properties of ConstrainedMentalClasses (p. 140) which can be used within mental (reasoning) processes of owning MentalSemiEntityTypes, i.e. pre- and post-conditions, commit conditions, cancel conditions and invariants. MentalConstraint, in addition to Constraint, allows specification of the kind of the constraint (for details see section *6.1.5 MentalConstraintKind,* p. 142).

MentalConstraints can be owned only by ConstrainedMentalClasses.

MentalConstraint, being a RedefinableElement, allows the redefinition of the values of constraint specifications (given by the specification meta-association inherited from UML Constraint), e.g. in the case of inherited owned ConstrainedMentalClasses, or redefinition specified by MentalProperties (p. 144). Specification of a redefined MentalConstraint is logically combined with the specification of the redefining MentalConstraint (of the same kind), following the rules specified in Tab. 6-1.

MentalConstraintKind	***Combination kind***
commitCondition	OR-ed
preCondition	OR-ed
invariant	Overridden
cancelCondition	OR-ed
postCondition	AND-ed

Tab. 6-1 Redefinition rules of MentalConstraints

Attributes

kind:MentalConstraint-Kind[1]	A kind of the MentalConstraint.

Constraints

1. The *commitPreCondition* literal cannot be the value of the kind meta-attribute.

   self.kind <> #commitPreCondition

Notation

In general, the MentalConstraint is depicted as a text string in braces ('{' '}') with the same format as defined in UML.

MentalConstraint can occur only within the context of an (a) owning ConstrainedMentalClass, or (b) owning MentalProperty. These metaclasses define specific notation for MentalConstraint and its placement.

Examples

Pre-condition of the DecidableGoal (p. 152) named InterceptBall (shown in Fig. 6-21) is specified as:

pre = {not self.team.hasBall(ball) and self.isFree()}

The pre-condition checks whether the robot is free and and its team already has the ball. The keyword 'self' is used to refer to an instance of the SoccerRobot AgentType (see Fig. 5-7 for details).

For other examples see Fig. 5-101, Fig. 6-21, Fig. 6-29, and Fig. 6-31.

Rationale

MentalConstraint is introduced to specify the properties of ConstrainedMentalClasses which can be used within mental (reasoning) processes of owning MentalSemiEntityTypes.

6.1.5 MentalConstraintKind

Semantics

MentalConstraintKind is an enumeration which specifies kinds of MentalConstraints (p. 141), as well as kinds of constraints specified for contributor and beneficiary in the Contribution relationship (p. 162).

If needed, the set of MentalConstraintKind enumeration literals can be extended.

Enumeration values

Tab. 6-2 specifies MentalConstraintKind's enumeration literals, keywords used for notation, and their semantics.

Value	*Keyword*	*Semantics*
commitCondition	commit	An assertion identifying the situation under which an autonomous entity commits to the particular ConstrainedMentalClass (if also the pre-condition holds).
preCondition	pre	The condition that must hold before the ConstrainedMentalClass can become effective (i.e. a goal can be committed to or a plan can be executed).

Tab. 6-2 MentalConstraintKind***'s enumeration literals***

Value	*Keyword*	*Semantics*
commitPreCondition	commpre	AND-ed combination of commitCondition and preCondition. Used only within Contribution.
invariant	inv	The condition that holds during the period the ConstrainedMentalClass remains effective.
cancelCondition	cancel	An assertion identifying the situation under which an autonomous entity cancels attempting to accomplish the ConstrainedMentalClass.
postCondition	post	The condition that holds after the Constrained-MentalClass has been accomplished (i.e. a goal has been achieved or a plan has been executed).

Tab. 6-2 MentalConstraintKind***'s enumeration literals***

Rationale MentalConstraintKind is introduced to specify the kinds of MentalConstraints and ends of a Contribution relationship.

6.1.6 MentalRelationship

Semantics MentalRelationship is an abstract specialized MentalState (p. 139), a superclass to all metaclasses defining the relationships between MentalStates.

There is one concrete subclass of the MentalRelationship—Contribution (p. 162).

Notation There is no general notation for MentalRelationship. The specific subclasses of MentalRelationship define their own notation.

Rationale MentalRelationship is introduced as a superclass to all metaclasses defining the relationships between MentalStates.

6.1.7 MentalSemiEntityType

Semantics MentalSemiEntityType is a specialized abstract Class (from UML), a superclass to all metaclasses which can own MentalProperties (p. 144), i.e. AutonomousEntityType (p. 24) and EntityRoleType (p. 39).

The ownership of a MentalProperty of a particular MentalClass (p. 140) type means that instances of the owning MentalSemiEntityType have control over instances of that MentalClass, i.e. they have (at least to some extent) a power or authority to manipulate those MentalClass instances (their decisions about those MentalClass instances are, to some degree, autonomous). For example, a MentalClass instance can decide:

- which Goal (p. 151) is to be achieved and which not,
- when and how the particular Goal instance is to be achieved,
- whether the particular Goal instance is already achieved or not,
- which Plan (p. 156) to execute, etc.

Instances of MentalSemiEntityTypes are referred to as *mental semi-entities*.

Associations

/mentalAttribute: MentalProperty[*]	A set of all MentalProperties owned by the MentalSemiEntityType. The association is ordered and derived. Subsets UML Class::ownedAttribute.

Constraints

1. The mentalAttribute meta-association refers to all ownedAttributes of the kind MentalProperty:

   ```
   mentalAttribute = self.ownedAttribute->
     select(oclIsKindOf(MentalProperty))
   ```

Notation There is no general notation for MentalSemiEntityType. The specific subclasses of MentalSemiEntityType define their own notation.

Rationale MentalSemiEntityType is introduced as a common superclass to all metaclasses which can own MentalProperties.

6.1.8 MentalProperty

Semantics MentalProperty is a specialized Property (from UML) used to specify that instances of its owner (i.e. mental semi-entities) have control over instances of the MentalClasses (p. 140) of its type, e.g. can decide whether to believe or not (and to what extent) in a Belief (p. 149), or whether and when to commit to a Goal (p. 151).

The attitude of a mental semi-entity to a belief or commitment to a goal is modeled by a Belief instance, or a Goal instance, being held in a slot of the corresponding MentalProperty.

The type of a MentalProperty can be only a MentalClass.

MentalProperties can be owned only by:

- MentalSemiEntityTypes (p. 143) as attributes, or
- MentalAssociations (p. 147) as member ends.

MentalProperties (except of MentalProperties of Belief type) can own MentalConstraints (p. 141) (each of a different type) to allow the redefinition of MentalConstraints of their types. The redefinition rules are described in section *6.1.4 MentalConstraint* (p. 141).

Note: The Plans (p. 156) controlled by MentalSemiEntityTypes are modeled as owned UML Activities, and therefore use of Plans as types of MentalProperties is forbidden, even if they are specialized MentalClasses.

Attributes

degree: ValueSpecification [0..1]	The degree of the MentalClass specified as the type of the MentalProperty. If specified, it overrides the degree value specified for the MentalClass itself.

Associations

association: MentalAssociation [0..1]	The MentalAssociation of which this MentalProperty is a member, if any. Redefines UML Property::association.
type: MentalClass[0..1]	The type of a MentalProperty. Redefines UML TypedElement::type.
mentalConstraint: MentalConstraint[*]	A set of MentalConstraints describing the MentalClass specified as the type. If specified, they redefine MentalConstraints specified for the MentalClass itself. Subsets UML Element::ownedElement.

Constraints

1. If the type meta-association is specified, the MentalClass referred to by it cannot be a Plan:

   self.type->notEmpty() implies (not self.type.oclIsKindOf(Plan))

2. Each mentalConstraint must have different kind:

   self.mentalConstraint->forAll(mc1, mc2 | mc1.kind<>mc2.kind)

3. The mentalConstraints can be specified only for a ConstrainedMentalClass:

   not (self.type->notEmpty() and
   self.type.oclIsKindOf(ConstrainedMentalClass)) implies
   self.mentalConstraint->isEmpty()

Notation

When shown as the end of a MentalAssociation, the MentalProperty is depicted as a UML association end, see section *6.1.9 MentalAssociation,* p. 147.

When shown as an attribute of an MentalSemiEntityType, the MentalProperty is depicted as a UML attribute with the stereotype <<mental>>. If specified, the value of the meta-attribute degree is depicted as a property string (tagged value) with the name ‘degree’. See Fig. 6-6.

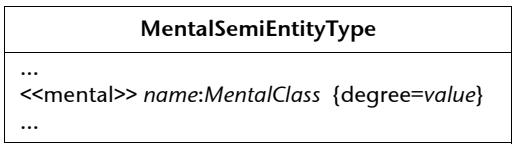


Fig. 6-6 Notation of MentalProperty shown as an attribute

Usually, the MentalConstraints of a MentalProperty are specified as hidden information, not shown in the diagram. However, if a user needs to express them explicitly, the MentalConstraints can be shown within a property string belonging to the MentalProperty. Each specified MentalConstraint has the form of a single tagged value following the format:

mental_constraint ::= *mental_constraint_kind* ‘=’ *specification*
mental_constraint_kind ::= ‘commit’ | ‘pre’ | ‘inv’ | ‘cancel’ | ‘post’

The *specification* is a specification of the meta-association of UML Constraint (i.e. a boolean expression), and *mental_constraint_kind* is a keyword identi-

fying the kind of the mental constraint (see section *6.1.5 MentalConstraint-Kind,* p. 142 for details).

Presentation options The MentalProperties of a MentalSemiEntityType can be placed in a special class compartment named <<mental>>. The stereotype <<mental>> of a particular MentalProperty is in this case omitted. See Fig. 6-7.

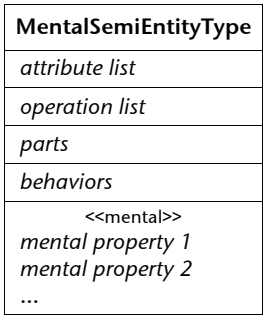


Fig. 6-7 Alternative notation of MentalProperties—placed in a common special class compartment

Another notational alternative is to group MentalProperties of one MentalSemiEntityType according to their fundamental types and to place each group into a specific class compartment. There are defined compartments <<beliefs>> for Beliefs, and <<goals>> for Goals. The stereotype <<mental>> of a particular MentalProperty is in this case omitted. See Fig. 6-8.

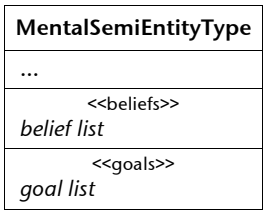


Fig. 6-8 Alternative notation of MentalProperties—grouped by fundamental types

Style guidelines MentalConstraints are usually specified as hidden information.

Examples Fig. 6-9 shows the definition of an EntityRoleType called Striker used to model a soccer striker, i.e. a player whose main job is to attack and try to score goals. The possibility to commit to these two goals is expressed by the MentalProperties named scoreGoal and attack, both of the fundamental type Goal. To commit to either of these goals, the Striker must believe that the goal is achievable. This is expressed by ownership of two MentalProperties named canScoreGoal and canAttack of the fundamental type Belief. To achieve the goals, the Striker defines two Plans called ScoreGoalPlan and AttackPlan.

Rationale MentalProperty is introduced to specify that mental semi-entities have control over Goal and Belief instances.

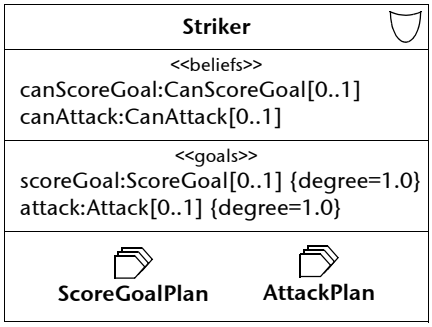


Fig. 6-9 Example of MentalProperty

6.1.9 MentalAssociation

Semantics MentalAssociation is a specialized Association (from UML) between a MentalSemiEntityType (p. 143) and a MentalClass (p. 140) used to specify a MentalProperty (p. 144) of the MentalSemiEntityType in the form of an association end.

MentalAssociation is always binary.

An instance of the MentalAssociation is called *mental link*.

Associations

memberEnd: Property[2]	Two associated Properties. This is an ordered association. Redefines UML Association::memberEnd.
mentalMemberEnd: MentalProperty[1]	Associated MentalProperty. Subsets MentalAssociation::memberEnd.

Notation MentalAssociation is depicted as a binary UML Association with the stereotype <<mental>>, see Fig. 6-10. The association end at the mentalMemberEnd's side can additionally show the value of its meta-attribute degree as a property string (tagged value) with the name 'degree'.

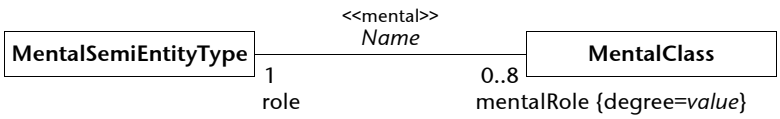


Fig. 6-10 Notation of MentalAssociation

Style guidelines Stereotype <<mental>> of the mentalMemberEnd is usually omitted.

The MentalAssociation's stereotype can be omitted from the diagram as well, i.e. an Association between MentalSemiEntityType and a MentalClass is considered to be a MentalAssociation, if it is evident from the context.

Examples Fig. 6-11 shows a diagram semantically equivalent to the diagram in Fig. 6-9, but all MentalProperties are depicted as MentalAssociations.

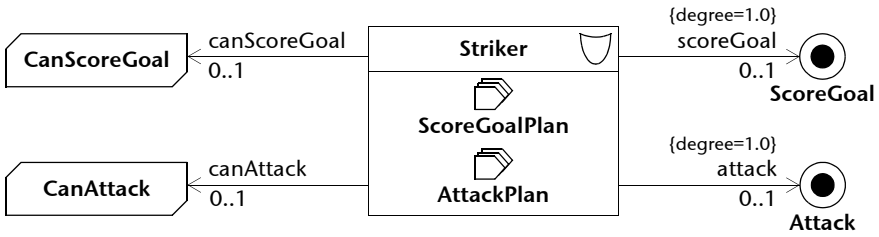


Fig. 6-11 Example of MentalAssociation

Rationale MentalAssociation is introduced to enable modeling of MentalProperties in the form of association ends. It is used to specify that mental semi-entities have control over Goal and Belief instances.

6.1.10 Responsibility

Semantics Responsibility is a specialized Realization (from UML) used to model a relation between MentalClasses (p. 140) (called *responsibility objects*) and NamedElements (from UML) (called *responsibility subjects*) that are obligated to accomplish (or to contribute to the accomplishment of) those MentalClasses (e.g. modification of Beliefs, p. 149, or achievement or maintenance of Goals, p. 151, or realization of Plans, p. 156).

Associations

subject: NamedElement[1..*]	The subject NamedElements responsible for object MentalClasses. Redefines UML Dependency::client.
object: MentalClass[1..*]	The set of MentalClasses the subject is responsible for. Redefines UML Dependency::supplier.

Notation Responsibility is depicted as a UML Realization (Dependency) relationship with the stereotype <<responsible>>. The responsibility subject (a NamedElement) represents a client and the responsibility object (a MentalClass) is a supplier. See Fig. 6-12.

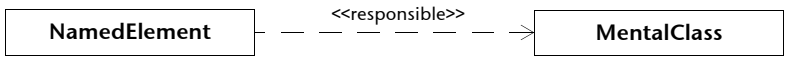


Fig. 6-12 Notation of Responsibility

Examples Fig. 6-13 shows an AgentType (p. 25) named Person with a Goal named StoreFluid. The ResourceTypes (p. 27) Cup, Bottle, and Glass can all realize that Goal and therefore are responsible for it. This responsibility is modeled by Responsibility relationships.

Rationale Responsibility is introduced to model which NamedElements are responsible for (or contribute to) the accomplishment of instances of which MentalClasses.

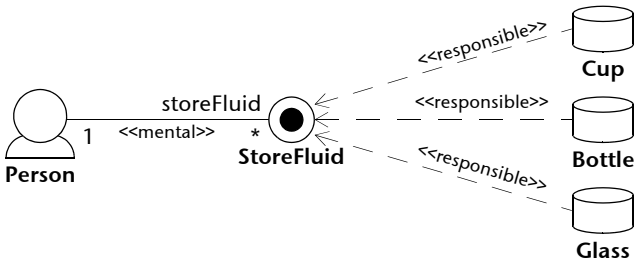


Fig. 6-13 Example of Responsibility

6.2 Beliefs

Overview The *Beliefs* package defines metaclasses used to model beliefs.

Abstract syntax The diagram of the Beliefs package is shown in Fig. 6-14.

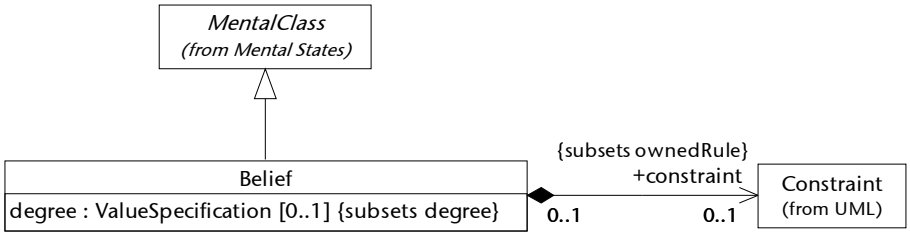


Fig. 6-14 Beliefs—belief

6.2.1 Belief

Semantics Belief is a specialized MentalClass (p. 140) used to model a state of affairs, proposition or other information relevant to the system and its mental model. If an instance of a Belief is held in a slot of a mental semi-entity's MentalProperty (p. 144), it represents the information which the mental semi-entity believes, and which does not need to be objectively true. The ability of a MentalSemiEntityType (p. 143) to believe in beliefs of a particular type is modeled by the ownership of a MentalProperty of the corresponding type.

The belief referred to by several mental semi-entities simultaneously represents their *common belief*.

The degree meta-association of a Belief specifies the reliability or confidence in the information specified by the Belief's constraint, i.e. a degree to which it is believed that the information specified by the Belief is true. AML does not specify either the syntax or semantics of degree's values, users are free to define and use their own. For example the values can be real numbers, integers, enumeration literals, expressions, etc.

The specification of the information a Belief represents is expressed by the owned Constraint (from UML).

When inherited, the owned constraint is overridden.

It is possible to specify attributes and/or operations for a Belief, to represent its parameters and functions, which can both be used in the owned constraint as static or computed values.

Attributes

degree: ValueSpecification [0..1]	Specification of the reliability or confidence in the information specified by the constraint meta-association, i.e. a degree to which the owning MentalSemiEntityType believes that the information specified by the Belief is true. Subsets MentalState::degree.

Associations

constraint: Constraint[0..1]	The specification of the information a Belief represents. Subsets UML Namespace::ownedRule.

Notation Belief is depicted as a UML Class with the stereotype <<belief>> and/or a special icon. If specified, the value of the meta-attribute degree is depicted as a property string (tagged value) with name 'degree', placed in the name compartment. The constraint, enclosed with braces ('{' '}'), is placed in a special class compartment. See Fig. 6-15.

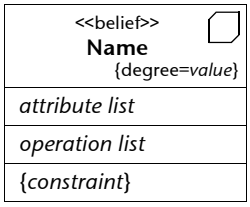


Fig. 6-15 Notation of Belief

Presentation options Belief can alternatively be depicted as a rectangle with beveled top-left and bottom-right corners. The stereotype icon and keyword is omitted in this case. See Fig. 6-16.

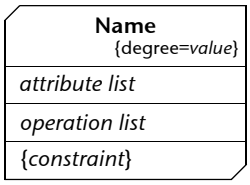


Fig. 6-16 Alternative notation of Belief—with a special shape

If not referred to explicitly in the model, the name of a Belief can be unspecified. In this case the name compartment of such a Belief can be omitted and only the constraint is specified. See Fig. 6-17.

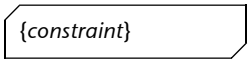


Fig. 6-17 Alternative notation of Belief—omitted name compartment

Style guidelines Owned operations, receptions, internal structure of parts and connectors, ports, supported and required interfaces, specification of owned behaviors, and nested classifiers are not usually specified for Beliefs.

Examples Fig. 6-18 shows a Belief called NearBall, which represents the belief that ball is nearby. The attribute near specifies what distance in meters is meant by "near". Its default value is 1.5, but can be changed at runtime. The constraint specifies that the distance from the ball is less or equal than the value of near.

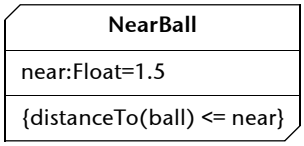


Fig. 6-18 Example of Belief

Rationale Belief is introduced to model beliefs.

6.3 Goals

Overview The *Goals* package defines metaclasses used to model goals.

Abstract syntax The diagram of the Goals package is shown in Fig. 6-19.

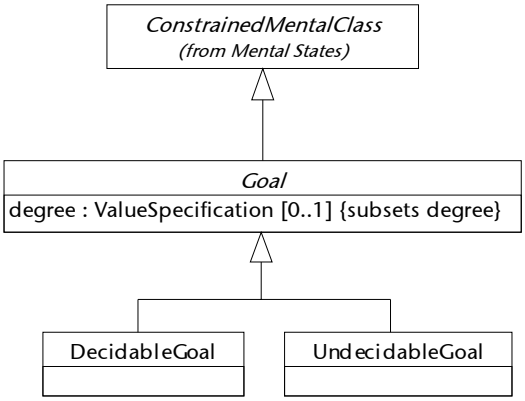


Fig. 6-19 Goals—goal hierarchy

6.3.1 Goal

Semantics Goal is an abstract specialized ConstrainedMentalClass (p. 140) used to model goals, i.e. conditions or states of affairs, with which the main concern is their achievement or maintenance. The Goals can thus be used to represent objectives, needs, motivations, desires, etc.

Commitment of a mental semi-entity to a goal is modeled by containment of the corresponding Goal instance by the value of the mental semi-entity's MentalProperty (p. 144).

The goal to which several mental semi-entities are committed to simultaneously represents their *common goal*.

The meta-attribute degree specifies the relative importance or appropriateness of the Goal. AML does not specify either the syntax or semantics of degree's values, users are free to define and use their own.

Goals can have attributes to specify parameters of their instances, e.g. the goal "Buy car" can have attributes carType, carColor, maxPrice, etc. Goals can have also operations to compute e.g. utility function(s) to determine how valuable the goal is, or operations computing the parameters of goals, etc.

Note: Different categories of goals used in goal-based requirements modeling approaches (e.g. KAOS [16], [70], NFR [11], [45], GBRAM [2])[8], can be specified for instance by special user-defined tagged values or by special naming conventions used for Goals, e.g. 'Maintain[Attack]', 'Achieve[ScoreGoal]', 'Avoid[ConcedeGoal]'. The goal categories may depend on the domain or methodology used, and therefore are not defined by AML. Users can define these themselves. See example in Fig. 6-23.

Attributes

degree: ValueSpecification [0..1]	The relative importance of the Goal. Subsets MentalState::degree.

Notation There is no general notation for Goal. The specific subclasses of Goal define their own notation.

Rationale Goal is introduced to define the common features of all its subclasses that are used to model concrete types of goals.

6.3.2 DecidableGoal

Semantics DecidableGoal is a specialized concrete Goal (p. 151) used to model goals for which there are clear-cut criteria according to which the goal-holder[9] can decide whether the DecidableGoal (particularly its postCondition; for details see section *6.1.5 MentalConstraintKind,* p. 142) has been achieved or not.

Note: DecidableGoal is also called 'hard goal' or simply 'goal' in some goal-based requirements modeling approaches (e.g. TROPOS [8], [68], i* [1], [79], [80], GRL [28], [42], KAOS [16], [70]).

Notation DecidableGoal is depicted as a UML Class with the stereotype <<dgoal>> and/or a special icon, see Fig. 6-20.

---

[8] Such as achievement, maintenance, avoidance, optimization, improvement, accuracy, etc.

[9] *Goal-holder* is the stakeholder or mental semi-entity which has control over a goal. Stakeholder is an individual who is affected by the system, or by the system modeling process.

If specified, the value of the meta-attribute degree is depicted as a property string (tagged value) with the name ‘degree’, placed in the name compartment.

The DecidableGoal rectangle can contain special compartments <<commit>>, <<pre>>, <<inv>>, <<cancel>>, and <<post>> for the contained MentalConstraints (p. 141). These compartments may be omitted and can be specified in any order.

If a DecidableGoal also specifies attributes, operations, behaviors, etc. they can be depicted in standard compartments as specified in UML.

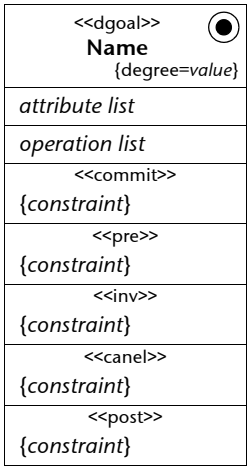


Fig. 6-20 Notation of DecidableGoal

Examples Fig. 6-21 shows the detail of a DecidableGoal named InterceptBall which represents a desire of a soccer robot to intercept the ball. Keyword ‘self’ from the owned MentalConstraints is therefore used to refer to the soccer robot (an instance of the SoccerRobot AgentType, see Fig. 5-7). The goal is committed to whenever the ball is near the robot. In order to commit to the goal, the robot must be free and its team cannot already have the ball. If the ball moves away from the robot, while trying to intercept the ball, the robot abandons this goal. The goal is successfully accomplished when the soccer robot obtains the ball. All aforementioned conditions are modeled as owned MentalConstraints.

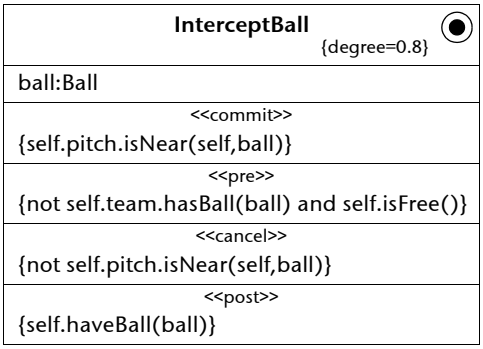


Fig. 6-21 Example of DecidableGoal

Rationale DecidableGoal is introduced to explicitly model decidable goals.

6.3.3 UndecidableGoal

Semantics UndecidableGoal is a specialized concrete Goal (p. 151) used to model goals for which there are no clear-cut criteria according to which the goal-holder can decide whether the postCondition (see section *6.1.5 MentalConstraintKind,* p. 142 for details) of the UndecidableGoal is achieved or not.

Note: UndecidableGoal is also called 'soft goal' or 'softgoal' in some goal-based requirements modeling approaches (e.g. TROPOS [8], [68], i* [1], [79], [80], GRL [28], [42], KAOS [16], [70], NFR [11], [45]).

Notation UndecidableGoal is depicted as a UML Class with the stereotype <<ugoal>> and/or a special icon, see Fig. 6-22.

If specified, the value of the meta-attribute degree is depicted as a property string (tagged value) with name 'degree', placed in the name compartment.

The UndecidableGoal rectangle can contain special compartments <<commit>>, <<pre>>, <<inv>>, <<cancel>>, and <<post>> for the contained MentalConstraints. These compartments may be omitted and can be specified in any order.

If an UndecidableGoal also contains attributes, operations, behaviors, etc. they can be depicted in standard compartments as specified in UML.

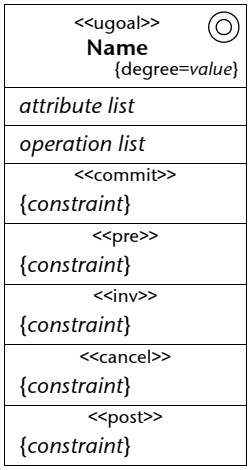


Fig. 6-22 Notation of UndecidableGoal

Examples Fig. 6-23 shows the fragment of a simple problem decomposition diagram for the development of a computer game. The diagram consists solely of UndecidableGoals, where each represents a non-functional requirement. Such diagrams can be used to describe system requirements and their relationships in the form of a mental model.

UndecidableGoals are decomposed using Contribution relationships (p. 162).

Classification of goals into categories (maintain, achieve, provide, or avoid) was accomplished by application of the naming convention (for de-

tails see section *6.3.1 Goal,* p. 151), where the category name precedes the goal's object placed in brackets.

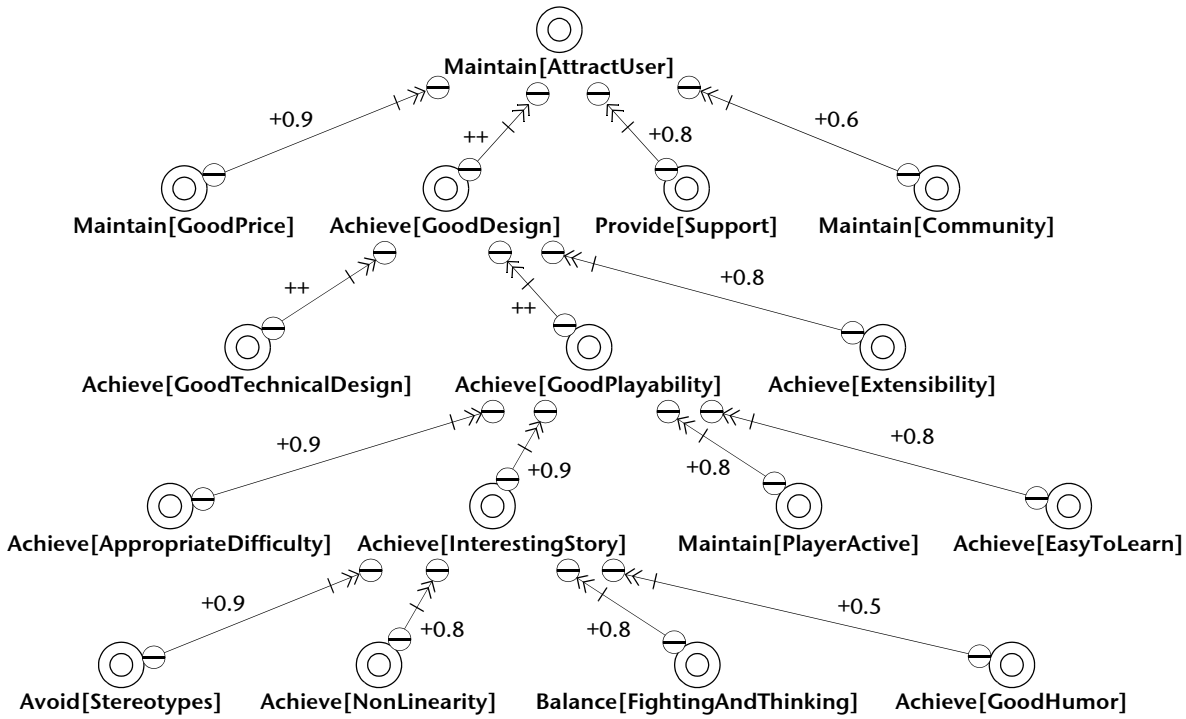


Fig. 6-23 Example of UndecidableGoal

Rationale UndecidableGoal is introduced to explicitly model undecidable goals.

6.4 Plans

Overview The *Plans* package defines metaclasses devoted to modeling plans.

Abstract syntax The diagrams of the Plans package are shown in figures Fig. 6-24 to Fig. 6-26.

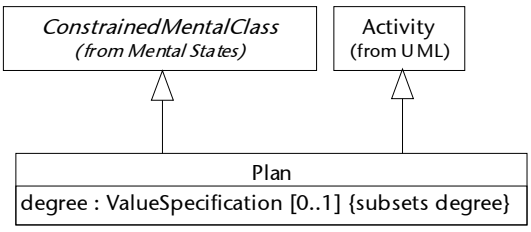


Fig. 6-24 Plans—plan

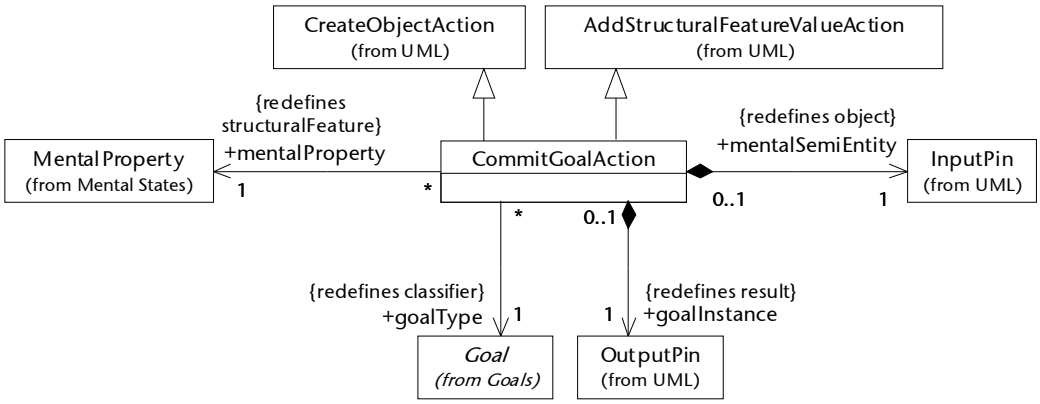


Fig. 6-25 Plans—commit goal action

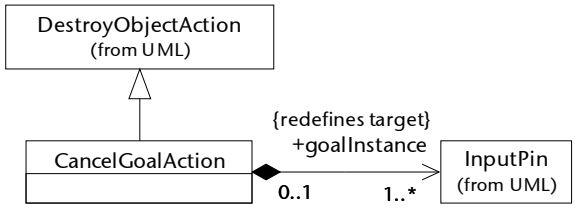


Fig. 6-26 Plans—cancel goal action

6.4.1 Plan

Semantics Plan is a specialized ConstrainedMentalClass (p. 140) and Activity (from UML), used to model capabilities of MentalSemiEntityTypes (p. 143) which represents either:

- predefined plans, i.e. kinds of activities a mental semi-entity's reasoning mechanism can manipulate in order to achieve Goals (p. 151), or
- fragments of behavior from which the plans can be composed (also called *plan fragments*).

In addition to UML Activity, Plan allows the specification of commit condition, cancel condition, and invariant (for details see section *6.1.5 MentalConstraintKind,* p. 142), which can be used by reasoning mechanisms[10].

For modeling the applicability of Plans, in relation to given Goals, Beliefs (p. 149) and other Plans, the Contribution relationship (p. 162) is used.

The meta-attribute degree specifies the relative preference of the Plan. AML does not specify either the syntax or semantics of degree's values, users are free to define and use their own.

[10] UML Activity specifies just pre-condition and post-condition.

Attributes

degree: ValueSpecification [0..1]	The relative preference of the Plan. Subsets MentalState::degree.

Constraints

1. Specification of the Constraint referred to by the precondition meta-association is identical with the specification of the MentalConstraint of kind *preCondition* referred to by the mentalConstraint meta-association, if the both are specified:

   self.precondition->notEmpty() and
   self.mentalConstraint->select(kind=#preCondition)->notEmpty() implies
   self.precondition.specification =
   self.mentalConstraint->select(kind=#preCondition).specification

2. Specification of the Constraint referred to by the postcondition meta-association is identical with the specification of the MentalConstraint of kind *postCondition* referred to by the mentalConstraint meta-association, if the both are specified:

   self.postcondition->notEmpty() and
   self.mentalConstraint->select(kind=#postCondition)->notEmpty()
   implies
   self.postcondition.specification =
   self.mentalConstraint->select(kind=#postCondition).specification

3. If the context (see UML Behavior::context meta-association) for Plan is specified, it must be a MentalSemiEntityType:

   self.context->notEmpty() implies
   self.context.oclIsKindOf(MentalSemiEntityType)

Notation

Plan is depicted as a UML Activity with the stereotype <<plan>> and/or a special icon, see Fig. 6-27.

If specified, the value of the meta-attribute degree is depicted as a property string (tagged value) with name 'degree', placed near the Plan name.

If the mental constraints of a Plan need to be displayed explicitly, they can be shown as stereotyped Constraints (from UML) placed into the Plan's rounded rectangle. For this, the stereotypes <<commit>>, <<pre>>, <<inv>>, <<cancel>>, and <<post>> are used.

If specified, UML standard Constraints precondition and postcondition, shown as stereotyped constraints <<precondition>> and <<postcondition>> are not shown because their values are identical to the MentalConstraints of kind *preCondition* and *postCondition*.

Style guidelines

Usually, the mental constraints of a Plan are specified as hidden information, and are not shown in the diagram.

Examples

See Fig. 5-101, Fig. 5-120, Fig. 6-29, and Fig. 6-31.

Rationale Plan is introduced to model predefined plans, or fragments of plans from which the plans can be composed.

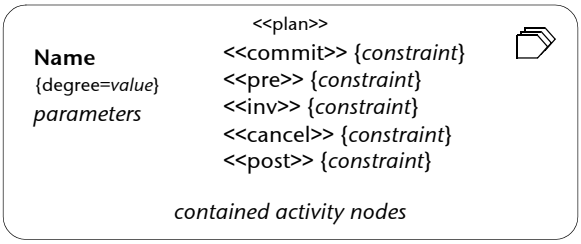


Fig. 6-27 Notation of Plan

6.4.2 CommitGoalAction

Semantics CommitGoalAction is a specialized CreateObjectAction (from UML) and AddStructuralFeatureValueAction (from UML), used to model the action of commitment to a Goal (p. 151).

This action allows the realization of the commitment to a Goal by instantiating the Goal and adding the created instance as a value to the MentalProperty (p. 144) of the mental semi-entity which commits to the Goal.

Commitment to an existing instance of a Goal can be modeled by AddStructuralFeatureValueAction (from UML) or by CreateLinkAction (from UML).

The CommitGoalAction specifies:

- what Goal is being instantiated (goalType meta-association),
- the Goal instance being created (goalInstance meta-association),
- the owning mental semi-entity committed to the Goal (mentalSemiEntity meta-association), and
- the MentalProperty, owned by the type of the owning mental semi-entity, to which the created Goal instance is added (mentalProperty meta-association).

If the MentalProperty referred to by the mentalProperty meta-association is ordered, the insertAt meta-association (inherited from the AddStructuralFeatureValueAction) specifies a position at which to insert the Goal instance.

Because the value meta-association (inherited from UML WriteStructuralFeatureAction) represents the same Goal instance as is already represented by the goalInstance meta-association, the properties of the InputPin referred to by the value meta-association are ignored in CommitGoalAction, and can be omitted in its specification.

Associations

goalType: Goal[1]	Instantiated Goal. Redefines UML CreateObjectAction::classifier.

goalInstance: OutputPin[1]	The OutputPin on which the created Goal instance is placed. Redefines UML CreateObjectAction::result.
mentalSemiEntity: InputPin[1]	The InputPin specifying the mental semi-entity committed to the Goal. Redefines UML StructuralFeatureAction::object.
mentalProperty: MentalProperty[1]	The MentalProperty where the created Goal instance is being placed. Redefines UML StructuralFeatureAction::structuralFeature.

Constraints

1. If the type of the InputPin referred to by the mentalSemiEntity meta-association is specified, it must be a MentalSemiEntityType:

   self.mentalSemiEntity.type->notEmpty() implies
   self.mentalSemiEntity.type.oclIsKindOf(MentalSemiEntityType)

2. If the type of the OutputPin referred to by the goalInstance meta-association is specified, it must conform to the Goal referred to by the goalType meta-association:

   self.goalInstance.type->notEmpty() implies
   self.goalInstance.type.conformsTo(self.goalType)

3. If the type of the MentalProperty referred to by the mentalProperty meta-association is specified, the Goal referred to by the goalType meta-association must conform to it:

   self.mentalProperty.type->notEmpty() implies
   self.goalType.conformsTo(self.mentalProperty.type)

4. CommitGoalAction can be performed only by a mental semi-entity:

   self.activity().hostClassifier().oclIsKindOf(MentalSemiEntityType)

Notation

CommitGoalAction is shown as a UML Action with the stereotype <<commit goal>> and/or a special icon, see Fig. 6-28.

Fig. 6-28 Notation of CommitGoalAction

Optionally, the name of the committing mental semi-entity, delimited by a period from the name of the MentalProperty referred to by the mentalProperty meta-association, may be specified in parentheses below the action's name. If the MentalProperty is ordered, the value of the insertAt meta-association can be placed after the MentalProperty's name in brackets.

If the committing mental semi-entity itself executes the CommitGoalAction, it can be identified by the keyword 'self' instead of committing mental semi-entity's name.

The created Goal instance is specified as an OutputPin. All notational variations for the UML OutputPin are allowed. The committed Goal is specified as the type of the OutputPin.

The mandatory InputPin referred to by the value meta-association has unspecified properties and is not drawn in diagrams.

Examples Fig. 6-29 shows the Plan (p. 156) of the EntityRoleType named Striker (see Fig. 6-11 for details) to kick a ball. If the striker is near the goal of the other team, it tries to kick towards the goal by committing to the Goal called KickGoal. If the striker is not near, it tries to find a teammate who is near and not offside. If such a teammate exists, the striker tries to pass the ball to the teammate by committing to the Goal called PassBall. If such a teammate does not exist, the striker simply tries to kick the ball away by committing to the Goal called KickBallAway.

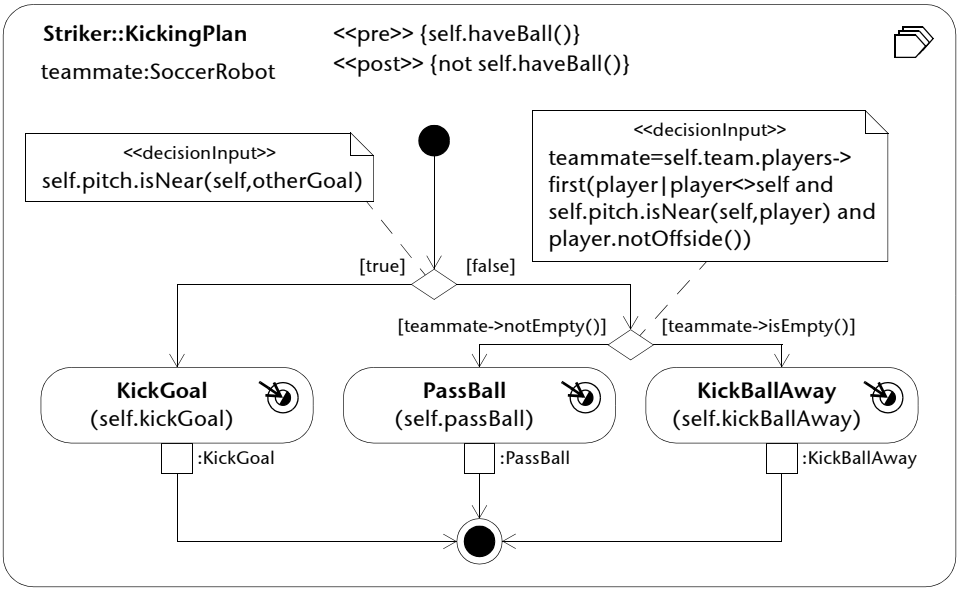


Fig. 6-29 Example of Plan and CommitGoalAction

Another example of CommitGoalAction is in Fig. 6-31.

Rationale CreateRoleAction is introduced to model commitment actions within Activities (Plans).

6.4.3 CancelGoalAction

Semantics CancelGoalAction is a specialized DestroyObjectAction (from UML) used to model de-commitment from goals.

This action allows the realization of de-commitment from a Goal (p. 151) by destruction of the corresponding Goal instance.

De-commitment from an instance of a Goal that does not need to be destroyed can be modeled by RemoveStructuralFeatureValueAction (from UML) or DestroyLinkAction (from UML).

Associations

goalInstance: InputPin[1..*]	The InputPins representing the Goal instances to be disposed. Redefines UML DestroyObjectAction::target.

Constraints

1. If the types of the InputPins referred to by the goalInstance meta-association are specified, they must be Goals:

   self.goalInstance->forAll(gi | gi.type.->notEmpty() implies gi.type.oclIsKindOf(Goal))

2. CancelGoalAction can be performed only by a mental semi-entity:

   self.activity().hostClassifier().oclIsKindOf(MentalSemiEntityType)

Notation CancelGoalAction is drawn as a UML Action with the stereotype <<cancel goal>> and/or a special icon, see Fig. 6-30. The cancelled Goal instances are depicted as InputPins.

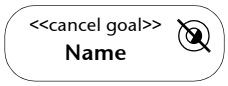


Fig. 6-30 Notation of CancelGoalAction

Examples Fig. 6-31 shows an overall Plan (p. 156) of the Striker's activity (for details see Fig. 6-11). It's main goal is to ScoreGoal and Attack. Both Goals are committed in parallel.

If the game is interrupted, the striker has to stop its activities, and therefore both the committed Goals are stopped, until the game continues.

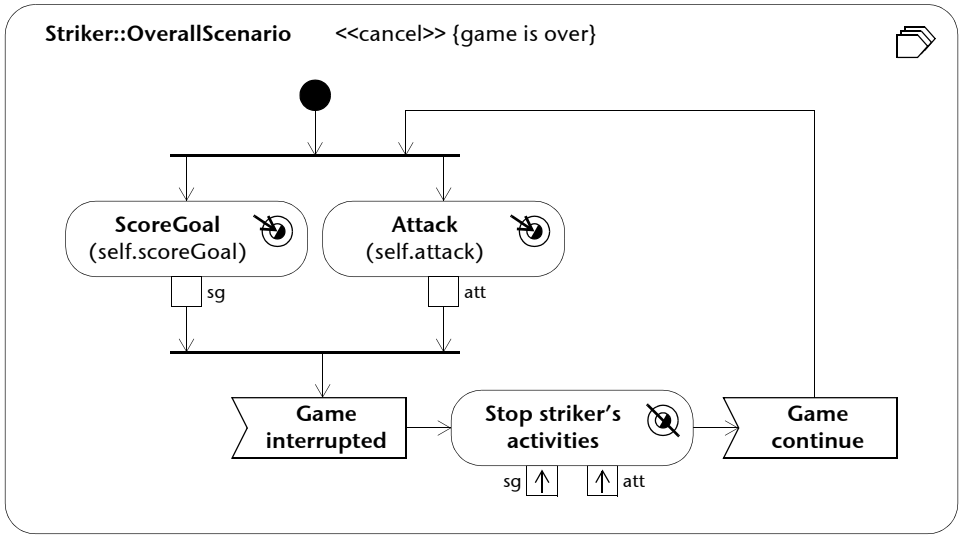


Fig. 6-31 Example of Plan , CommitGoalAction, and CancelGoalAction

Rationale CancelGoalAction is introduced to model de-commitment to goals.

6.5 Mental Relationships

Overview The *Mental Relationships* package defines metaclasses used to model relationships between MentalStates (p. 139) which can support reasoning processes.

Abstract syntax The diagram of the Mental Relationships package is shown in Fig. 6-32.

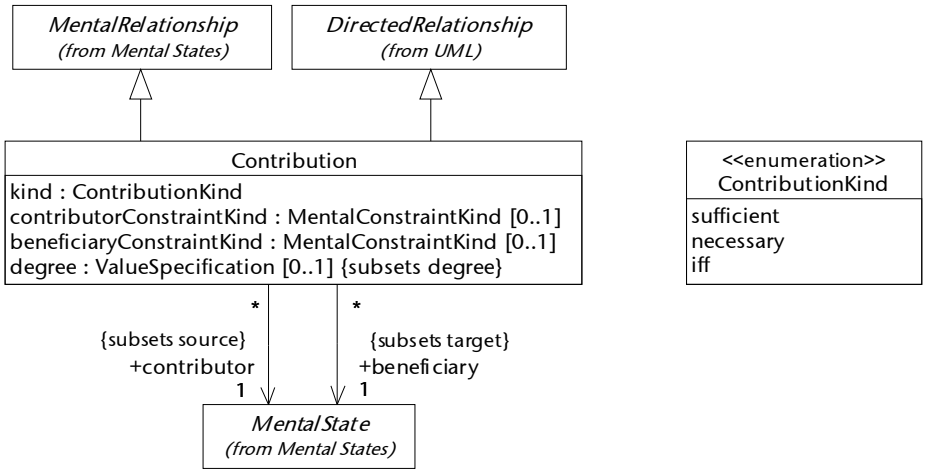


Fig. 6-32 Mental Relationships—contribution

6.5.1 Contribution

Semantics Contribution is a specialized MentalRelationship (p. 143) and DirectedRelationship (from UML) used to model logical relationships between MentalStates (p. 139) and their MentalConstraints (p. 141).

The manner in which the *contributor* of the Contribution relationship (i.e. a MentalState referred to by the contibutor meta-association) influences its *beneficiary* (i.e. a MentalState referred to by the beneficiary meta-association) is specified by values of meta-attributes of the particular Contribution.

The meta-attribute kind determines whether the contribution of the contributor's MentalConstraint of a given kind (specified by the meta-attribute contributorConstraintKind) is a necessary, sufficient, or equivalent condition for the beneficiary's MentalConstraint of a given kind (specified by the meta-attribute beneficiaryConstraintKind).

The meta-attribute contributorConstraintKind specifies the kind of a MentalConstraint of the contributor which contributes in some way to a kind of MentalConstraint of the beneficiary, specified by the beneficiaryConstraintKind meta-attribute. For example, a Contribution can specify that a postcondition of the contributor contributes in some way (e.g. in a positive and sufficient way) to the precondition of the related beneficiary. For details about possible values of the constraint kinds see section *6.1.5 MentalConstraintKind,* p. 142.

If contributor and/or beneficiary is a Belief (p. 149), the contributorConstraintKind and/or the beneficiaryConstraintKind meta-attribute is unspeci-

fied. In this case the Belief's constraint is considered to contribute or benefit.

If the contributor and/or beneficiary is a Contribution, the contributorConstraintKind and/or the beneficiaryConstraintKind meta-attributes are also unspecified.

The meta-attribute degree can be used to specify the extent to which the contributor influences the beneficiary. AML does not specify either the syntax or semantics of degree's values, users are free to define and use their own.

Attributes

kind: ContributionKind[1]	Determines whether the contribution of the contributor's MentalConstraint of a specified kind is a necessary, sufficient, or equivalent condition for the beneficiary's MentalConstraint of a specified kind.
contributorConstraintKind:MentalConstraintKind[0..1]	The kind of the contributor's MentalConstraint which contributes to the kind of the beneficiary's MentalConstraint (specified by the beneficiaryConstraintKind meta-attribute).
beneficiaryConstraintKind:MentalConstraintKind[0..1]	The kind of the beneficiary's MentalConstraint to which the kind of the contributor's MentalConstraint (specified by the contributorConstraintKind meta-attribute) contributes.
degree: ValueSpecification [0..1]	Degree of influence. Subsets MentalState::degree.

Associations

contributor: MentalState[1]	The contributor of the Contribution. Subsets UML DirectedRelationship::source.
beneficiary: MentalState[1]	The beneficiary of the Contribution. Subsets UML DirectedRelationship::target.

Constraints

1. If the MentalState referred to by the contributor meta-association is a Belief or a Contribution, the contributorConstraintKind meta-attribute is unspecified:

   self.contributor.oclIsKindOf(Belief) or
   self.contributor.oclIsKindOf(Contribution) implies
   self.contributorConstraintKind->isEmpty()

2. If the MentalState referred to by the beneficiary meta-association is a Belief or a Contribution, the beneficiaryConstraintKind meta-attribute is unspecified:

```
self.beneficiary.oclIsKindOf(Belief) or
self.beneficiary.oclIsKindOf(Contribution) implies
self.beneficiaryConstraintKind->isEmpty()
```

Notation Sufficient Contribution (i.e. kind=#sufficient) is depicted as an arrow with double arrowhead, leading from the contributor to the beneficiary. See Fig. 6-33.

Contribution's degree is depicted as a label (usually an expression) placed near the arrow line.

The value of the contributorConstraintKind is depicted as a keyword placed near the Contribution's tail. The value of the beneficiaryConstraintKind is depicted as a keyword placed near the Contribution's arrowhead. The keywords are specified in section *6.1.5 MentalConstraintKind* (p. 142).

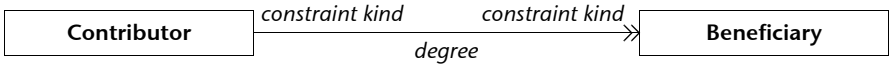


Fig. 6-33 Notation of sufficient Contribution

Necessary Contribution (i.e. kind=#necessary) is depicted as the sufficient Contribution, but the arrow line is crossed by a short line near the arrowhead. See Fig. 6-34.

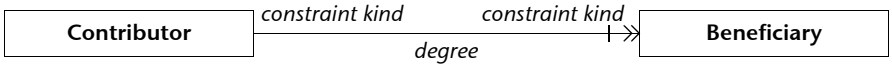


Fig. 6-34 Notation of necessary Contribution

Equivalence (iff) Contribution (i.e. kind=#iff) is depicted by a line with arrowheads placed at each end. See Fig. 6-35.

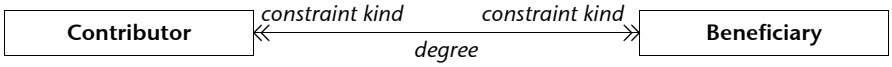


Fig. 6-35 Notation of equivalence (iff) Contribution

Presentation options A graphical icon for particular MentalConstraintKind can be used instead of textual labels. Alternative notation is depicted in Fig. 6-36.

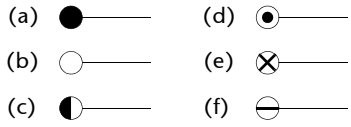


Fig. 6-36 Alternative notation for MentalConstraintKind used for Contribution ends: *(a) commitCondition, (b) preCondition, (c) commitPreCondition, (d) invariant, (e) cancelCondition, and (f) postCondition.*

Examples ***Contribution as a logical implication.*** Contribution can be understood as a logical implication between MentalStates or their MentalConstraints (if

specified). Tab. 6-3 provides interpretation of different Contribution kinds using the logical implication.

Contribution	*Logic interpretation*	*Description*
C ⟶» B	C ⇒ B	C is sufficient for B
C ⟶|» B	C ⇐ B	C is necessary for B
C «⟶» B	C ⇔ B	C is equivalent with B (C is sufficient and necessary for B)

Tab. 6-3 Logic interpretation of Contribution. *C stands for contributor, B for beneficiary, ⇒ denotes logical implication, and ⇔ logical equivalence.*

> ***Note:*** In this interpretation, the sufficient Contribution leading from A to B is logically equal to a necessary Contribution leading from B to A. Even though this property of strictly logical interpretation may tempt one to use just one kind of Contribution in models, from the domain perspective it may be required to use both, sufficient and necessary, Contribution kinds to express the mental model in a more natural and comprehensive way.

Combination of sufficient and necessary Contributions. If a set of Contributions of the same kind have the same beneficiary and the value of their beneficiaryConstraintKind meta-attribute (if specified), their logical interpretations can be combined according to the following rules:

- contributors of all sufficient Contributions are logically OR-ed to form the antecedent[11] of the resulting implication,
  i.e. $(contributor_1 \vee ... \vee contributor_n) \Rightarrow beneficiary$, and
- contributors of all necessary Contributions are logically AND-ed to form the consequent[11] of the resulting implication,
  i.e. $beneficiary \Rightarrow (contributor_1 \wedge ... \wedge contributor_n)$.

Operator $\vee$ stands for logical OR (disjunction), and $\wedge$ represents logical AND (conjunction).

According to the aforementioned rules, the model presented in Fig. 6-37 could be interpreted as follows:

P.pre ⇒ (G1.commitPre ∧ G2.commitPre)
(G3.post ∨ B) ⇒ P.post

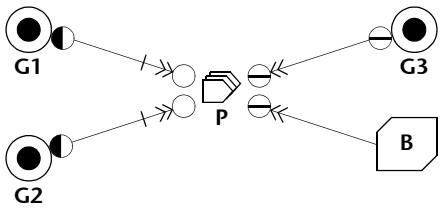


Fig. 6-37 Example of combining sufficient and necessary Contributions

[11] The *antecedent* is the operand of the implication which represent the sufficient condition, and the *consequent* is the operand which represent the necessary condition, i.e. *antecedent* => *consequent*.

Complex logical relationships. Contributions in combination with Beliefs can also be used to express more complex logical expressions between MentalStates.

Fig. 6-38 shows three semantically equivalent model variants, that could be interpreted as the following logical formula:

$$((G1.post \wedge G2.post) \vee B) \Rightarrow P.pre$$

The expression G1.post denotes postcondition of DecidableGoal (p. 152) G1, G2.post postcondition of the DecidableGoal G2, B1 represents constraints of the Belief B, and P.pre stands for precondition of Plan (p. 156) P1.

The example shows a usage of Beliefs for modeling complex conditions and their possible decomposition. Presented model variants show different levels of formula decomposition and modeling of constituents explicitly.

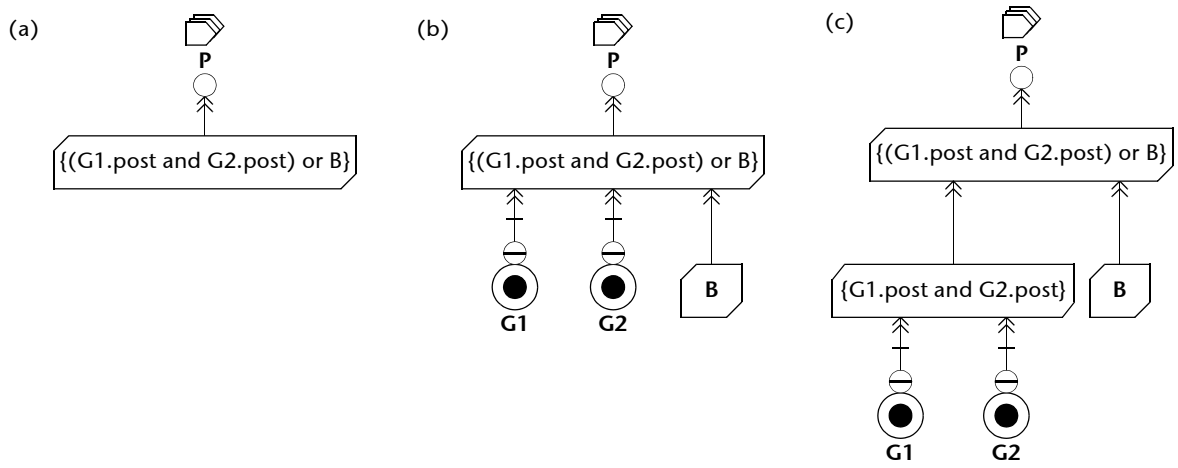


Fig. 6-38 Example of modeling complex logical relationships between MentalStates: *(a) no further decomposition of the contributing Belief, (b) a simple one-level depth decomposition of the contributing Belief, and (c) a full decomposition of the contributing Belief into its constituents modeled explicitly.*

Examples of notation and semantics for degree. The degree meta-attribute can be specified as a ***numeric value***—a real number from interval <-1,1>. Positive numbers represent a *positive contribution* of the contribution to the beneficiary, negative numbers represent a *negative contribution*. The larger the number is, the more degree of contribution it represents. Zero value represents an *indifferent contribution*, i.e. the contributor does not influence the beneficiary at all.

Another method of specifying the value of the degree is by using ***symbolic literals*** with predefined semantics. Tab. 6-4 provides examples of such literals, their interpretation and mapping to numeric values.

The both forms can be combined in one model.

Value		***Interpretation***	
Symbolic	***Numeric***	***Intuitive***	***Modal logic***
(empty)		A implies C	$A \Rightarrow C$
0	0	C is indifferent to A	$A \Rightarrow \Box(C \vee \neg C)$
+	+0.5	A contributes to C	$A \Rightarrow \Diamond C$
++	+1.0	A contributes to C strongly	$A \Rightarrow \Box C$
-	-0.5	A conflicts with C	$A \Rightarrow \Diamond \neg C$
--	-1.0	A conflicts with C strongly	$A \Rightarrow \Box \neg C$

Tab. 6-4 Examples of values of the Contribution's degree. *A indicates implication antecedent and C stands for implication consequent (i.e. $A \Rightarrow C$). The implication here represents the logic interpretation of a Contribution, as defined before. The modal operator $\Diamond$ stands for "possible", and operator $\Box$ for "necessary". See [75] for details. Symbol $\neg$ denotes logical negation.*

> ***Note:*** The formula $\Box A \Rightarrow A$ is an axiom (known also as the *axiom T*) of the basic modal logics (KT, KT4, KT5, etc.). If such logics are used to interpret the modal operators, the degrees *implies* (empty string) and *contributes strongly* ('++') are semantically equivalent.

Advanced schemas for the degree can also use more complex expressions.

Contribution as contributor or beneficiary. The Contribution relationship can be the contributor as well as the beneficiary of another Contribution. Fig. 6-39 shows an example in which achievement of the DecidableGoal FillCoffeeMachine contributes to the precondition of the DecidableGoal PrepareCoffee, but only if the coffee machine is not damaged. Furthermore, if preparation of the coffee (by accomplishment of the PrepareCoffee DecidableGoal) causes it to be drunk (by committing to the Drink DecidableGoal, which represents drinking of a prepared beverage), it is deduced that a person committed to these goals may like coffee.

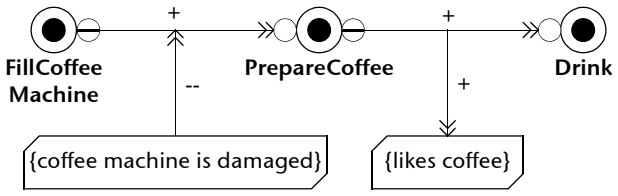


Fig. 6-39 Example of contribution to Contribution, and contributing Contribution

Real-world examples. Fig. 6-40 shows a causal analysis of the Plan PassBallPlan. The diagram depicts the necessary conditions for the Plan execution, as well as the consequences of its successful accomplishment.

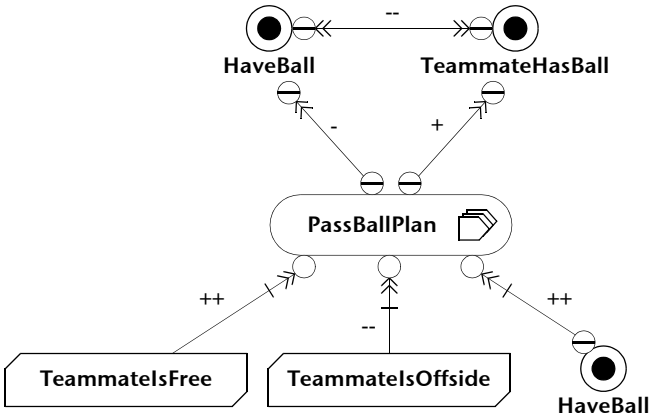


Fig. 6-40 Example of Contribution used in causal analysis

The diagram from Fig. 6-40 should be interpreted as a set of the following logical formulas:

PassBallPlan.pre ⇒ □TeammateIsFree
PassBallPlan.pre ⇒ □¬TeammateIsOffside
PassBallPlan.pre ⇒ □HaveBall.post
PassBallPlan.post ⇒ ◇¬HaveBall.post
PassBallPlan.post ⇒ ◇TeammateHasBall.post
HaveBall.post ⇒ □¬TeammateHasBall.post
TeammateHasBall.post ⇒ □¬HaveBall.post

Fig. 6-41 shows an example of the problem decomposition model aimed at specifying how to win a soccer game. The main DecidableGoal of a soccer player WinGame is decomposed into other DecidableGoals and Beliefs by means of Contribution relationships.

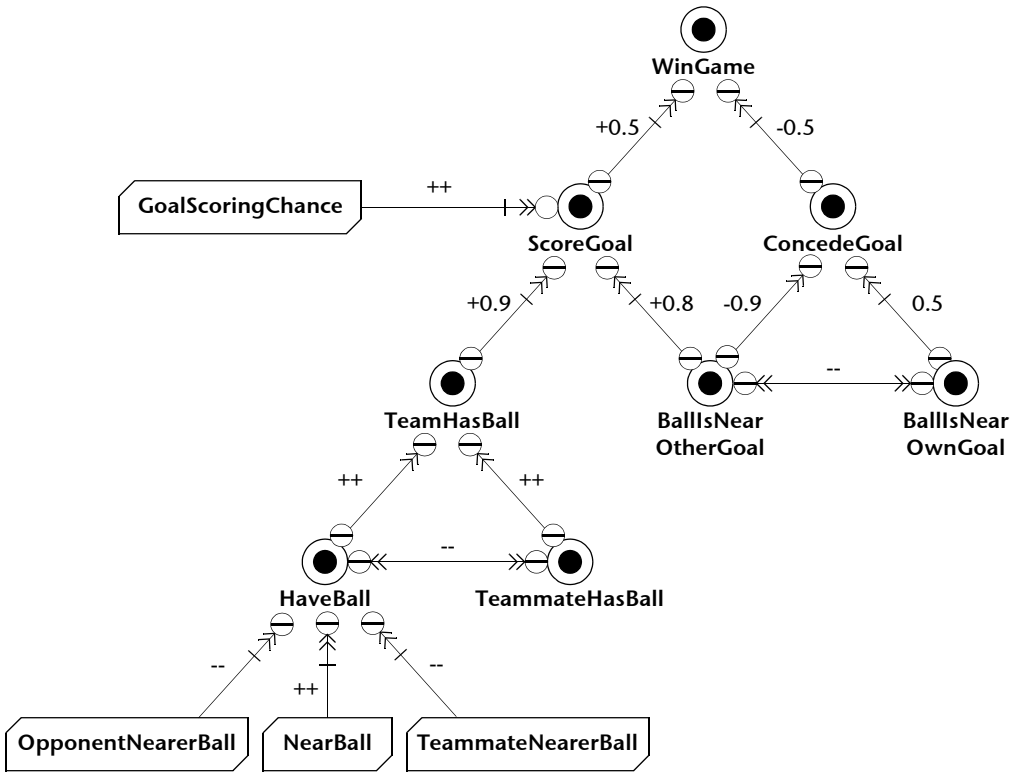


Fig. 6-41 Example of Contribution used for problem decomposition analysis

Indifferent Contribution is used to explicitly model independence of MentalStates and their MentalConstraints. Fig. 6-42 shows an example which specifies that scoring a goal (specified by the ScoreGoal's post-condition) is independent of the Belief that the player is younger than 18 years.

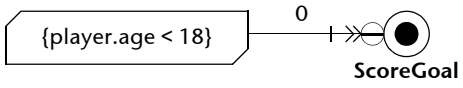


Fig. 6-42 Example of indifferent Contribution

Another example of the Contribution can be found in Fig. 6-23.

Rationale Contribution is introduced to model logical relationships between MentalStates and their MentalConstraints.

6.5.2 ContributionKind

Semantics ContributionKind is an enumeration which specifies possible kinds of Contributions.

AML supports sufficient, necessary and equivalent (iff) contribution kinds. If needed, the set of ContributionKind enumeration literals can be extended.

Enumeration values Tab. 6-5 specifies ContributionKind's enumeration literals and their semantics.

Value	***Semantics***
sufficient	The contributor or its MentalConstraint (p. 141) of the given kind (if specified) is a sufficient condition for the beneficiary or its MentalConstraint of the given kind (if specified).
necessary	The contributor or its MentalConstraint of the given kind (if specified) is a necessary condition for the beneficiary or its MentalConstraint of the given kind (if specified).
iff	(if and only if) The contributor and beneficiary or their MentalConstraints of the given kinds (if specified) are equivalent, i.e. the contributor or its MentalConstraint of the given kind (if specified) is a sufficient and necessary condition for the beneficiary or its MentalConstraint of the given kind (if specified).

Tab. 6-5 ContributionKind's enumeration literals

Rationale ContributionKind is introduced to define possible kinds of Contributions.

7 Contexts

Overview The *Contexts* package defines the metaclasses used to logically structure models according to situations that can occur during a system's lifetime and to model elements involved in handling those situations.

Abstract syntax The diagram of the Contexts package is shown in Fig. 7-1.

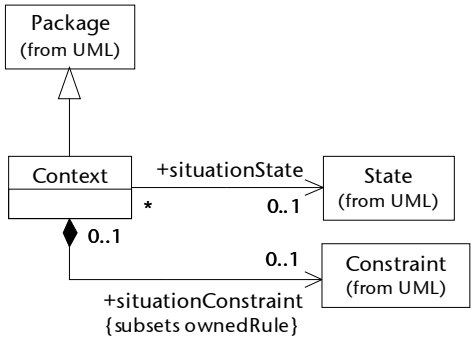


Fig. 7-1 Contexts—context

7.0.1 Context

Semantics Context is a specialized Package (from UML) used to contain a part of the model relevant for a particular situation. The situation is specified either as a Constraint (from UML) or an explicitly modeled State (from UML) associated with the Context.

Associations

situationState: State[0..1]	The State determining the situation for the Context.
situationConstraint: Constraint[0..1]	The Constraint determining the situation for the Context. Subsets UML Namespace::ownedRule.

Constraints

1. Either the situationState or the situationConstraint meta-association can be specified:

   self.situationState->notEmpty() xor self.situationContext->notEmpty()

Notation Context is depicted as a UML Package with the stereotype <<context>> and/or a special icon, see Fig. 7-2.

The name of the State referred to by the situationState meta-association, or the specification of the Constraint referred to by the situationConstraint meta-association may be placed after or below the Context's name.

If the situation is specified as a State, it uses the following format:

'[' *state_name* ']'

The *state_name* represents the name of the State referred.

If the situation is specified as a Constraint, it uses the syntax of Constraint as defined in UML—a text string in braces (‘{’ ‘}’).

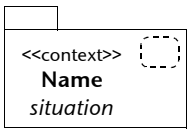


Fig. 7-2 Notation of Context

Presentation options Context can also be depicted as a dashed large rounded rectangle with a small rectangle (a “tab”) attached to the top left side of the large rounded rectangle.

If the members of the Context are not shown, the name and the situation is placed within the large rounded rectangle, see Fig. 7-3 (a).

If the members of the Context are shown within the large rounded rectangle, the name and the situation is placed within the tab, see Fig. 7-3 (b). The visibility of a Context member may be indicated as for UML Package. Members may also be shown by branching lines (“nesting” relationship) to member elements, drawn outside the package, for details see [53].

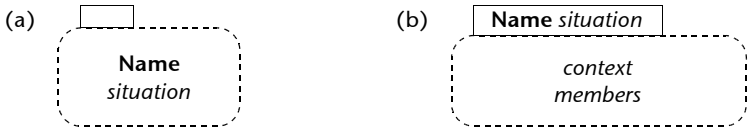


Fig. 7-3 Alternative notations of Context: *(a) members are hidden, (b) members are shown.*

Examples Fig. 7-4 shows the Context used to describe the substitution of soccer players. It contains the algorithm of substitution, interaction of entities taking part in substitution, as well as definition of BehaviorFragments (p. 65) defining the substitution-specific Capabilities (p. 62) of affected EntityRoleTypes (p. 39).

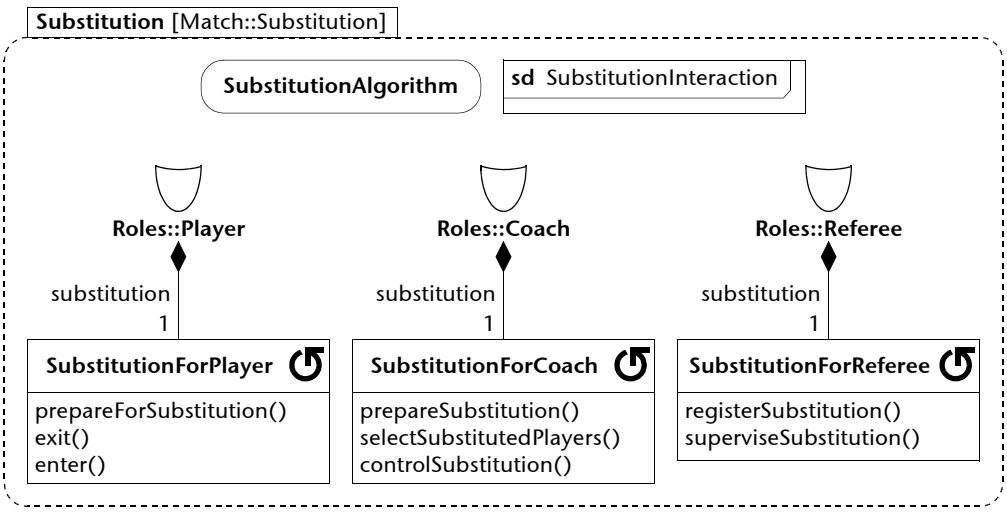


Fig. 7-4 Example of defining Context

The Usage and ElementImport relationships (both from UML) can be used to model participation of elements in Contexts. Fig. 7-5 demonstrates application of the Usage relationships to depict dependency of EntityRoleTypes on particular Contexts. Each Context may define the Capabilities, Interactions, provided and used services, StateMachines, Activities, social relationships, etc. of the related EntityRoleTypes, specified for the referred situation. The EntityRoleTypes (and possibly also other modeling elements) are then composed of situation-specific parts. This kind of situation-based decomposition enables to specify complex behavior and structural features of modeling elements in a flexible and comprehensive way.

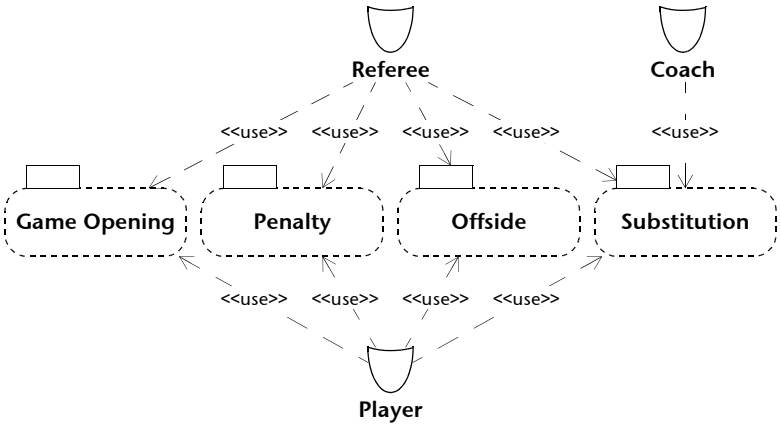


Fig. 7-5 Example of using Contexts

Rationale Context is introduced to offer the possibility to logically structure models according to the situations which can occur during a system's lifetime and to model elements involved in handling those situations.

8 UML Extension for AML

Overview The *UML Extension for AML* package adds the meta-properties defined in the AML Kernel package to the standard UML 2.0 Superstructure metaclasses. It is a non-conservative extension of UML, and is an optional part of the language.

Abstract syntax The diagram of the UML Extension for AML package is shown in Fig. 8-1.

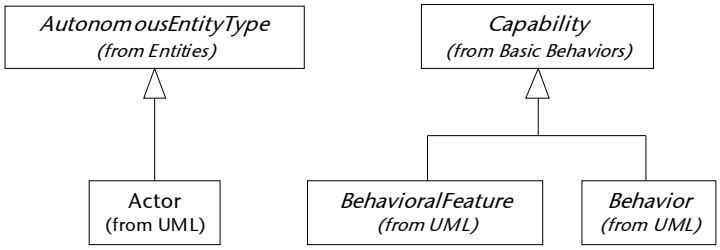


Fig. 8-1 UML Extensions for AML—UML element extensions

8.0.1 Extended Actor

Semantics Actor, being a specialized AutonomousEntityType (p. 24) and an extension of UML Actor, can:

- own MentalProperties (p. 144),
- have Capabilities (p. 62),
- be decomposed into BehaviorFragments (p. 65),
- provide and/or use services (see section *5.4 Services,* p. 101),
- observe and/or effect its environment (see section *5.5 Observations and Effecting Interactions,* p. 116),
- play entity roles (see section *4.5 Social Aspects,* p. 30),
- participate in social relationships (see section *4.5 Social Aspects,* p. 30), and
- specify values of the meta-attributes defined by the SocializedSemiEntityType (p. 33).

Examples Fig. 8-2 shows an Actor called Player which represents a user of an RPG (Role Playing Game) computer game. Its goal, to win the game, is modeled by a MentalAssociation (p. 147) with the WinGame DecidableGoal (p. 152). Additionally, the Player can either play an entity role of a hero (modeled by

the Hero EntityRoleType, p. 39) or a creature (modeled by the Creature EntityRoleType).

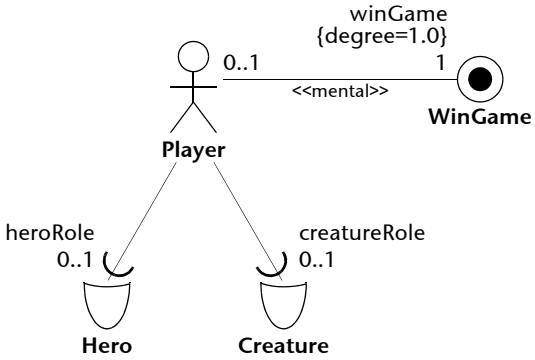


Fig. 8-2 Example of extended Actor

Rationale Extension of UML Actor is introduced to allow the modeling of Actors as AutonomousEntityTypes.

8.0.2 Extended BehavioralFeature

Semantics BehavioralFeature, being a specialized Capability (p. 62), can in addition to UML BehavioralFeature also specify meta-associations: inputs, outputs, pre-conditions, and post-conditions.

Notation See section *5.1.2 Capability,* p. 62.

Examples The Operation[12] (from UML) shoot() of the SoccerRobot AgentType (p. 25) specifies pre- and post-conditions, see Fig. 8-3. The pre-condition constrains the soccer robot invoking the Operation such that it must have the ball, and the post-condition specifies that after successful execution of the shoot() Operation, the soccer robot no longer possesses the ball and the ball will have changed its position.

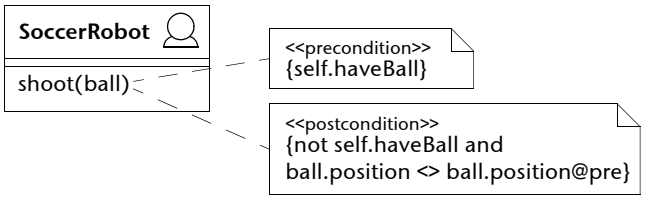


Fig. 8-3 Example of extended BehavioralFeature

Rationale The extension of BehavioralFeature is introduced to unify common meta-attributes of BehavioralFeature and Behavior in order to refer to them uniformly e.g. while reasoning.

[12] Operation is a specialized BehavioralFeature.

8.0.3 Extended Behavior

Semantics Behavior, being a specialized Capability (p. 62), can in addition to UML Behavior also specify meta-associations: inputs, outputs, pre-conditions, and post-conditions.

Notation See sections *5.1.2 Capability,* p. 62 and *6.4.1 Plan,* p. 156.

Examples The Activity (from UML) called SubstitutionAlgorithm, shown also in Fig. 7-4, can specify the pre- and post-conditions, see Fig. 8-4.

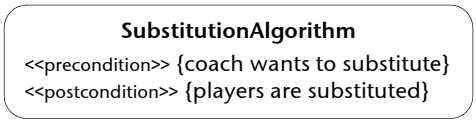


Fig. 8-4 Example of extended Behavior

For other examples see Fig. 5-101 and Fig. 6-29.

Rationale Extension of Behavior is introduced to unify common meta-attributes of BehavioralFeature and Behavior in order to refer to them uniformly e.g. while reasoning.

9 Diagrams

9.1 Diagram Frames

AML extends the UML 2.0 notation of the diagram frames by:

- an alternative syntax of the heading of the diagram frame, and
- the possibility to explicitly specify the list of template parameters for diagram frames which represent templates.

Fig. 9-1 depicts the notation of the AML diagram frame.

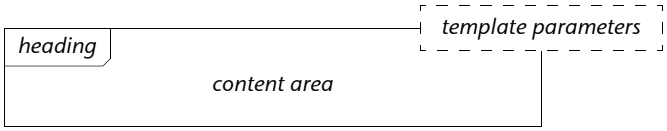


Fig. 9-1 Diagram frame notation

Heading The heading of the diagram frame has the following syntax:

heading ::= [*kind*][*owner*]['::'*diagram-name*][*property-string*]

The *kind* is the type and the *owner* is the name (also possibly containing parameters, the type of the return value, binding information, etc.) of the namespace enclosing, or the model element owning, elements in the diagram (as defined by UML [53]). The *diagram-name* is the name of the diagram and the *property-string* specifies the tagged values of the namespace enclosing, or the model element owning, elements in the diagram.

If needed, the set of MentalConstraintKind enumeration literals can be extended.

AML extends the set of UML diagram frame kinds, see Tab. 9-1. If needed, this set can be extended also by other diagram frame kinds (e.g. for new modeling elements added by users).

Name	***Short form***	***Type of owner***
agent		AgentType (p. 25)
resource	res	ResourceType (p. 27)
environment	env	EnvironmentType (p. 28)
organization unit	orgu	OrganizationUnitType (p. 31)
role		EntityRoleType (p. 39)
agent execution environment	aee	AgentExecutionEnvironment (p. 49)
ontology	ont	Ontology (p. 56)
ontology class	oclass	OntologyClass (p. 57)
behavior fragment	bf	BehaviorFragment (p. 65)
interaction protocol	ip	InteractionProtocol (p. 89)
service specification	sspec	ServiceSpecification (p. 103)

Tab. 9-1 AML-specific diagram kinds

Name	*Short form*	*Type of owner*
service protocol	sp	ServiceProtocol (p. 105)
perceptor type	pct	PerceptorType (p. 118)
effector type	eft	EffectorType (p. 124)
plan		Plan (p. 156)
context	ctx	Context (p. 171)
actor		Actor (from UML)

Tab. 9-1 AML-specific diagram kinds

Template parameters AML frames representing the templates (parametrized elements) can explicitly specify the list of TemplateParameters (from UML) of the represented element. The list of TemplateParameters is placed in a dashed rectangle in the upper right corner of the diagram frame.

To enable logical grouping of the TemplateParameters, their subsets can be depicted in the form of stereotyped lists, or placed into separate compartments. Both possible notations are depicted in Fig. 9-2. The stereotypes of parameter lists (i.e. <<stereotype X>> and <<stereotype Y>> in the figure) identify application of the subsequent TemplateParameters.

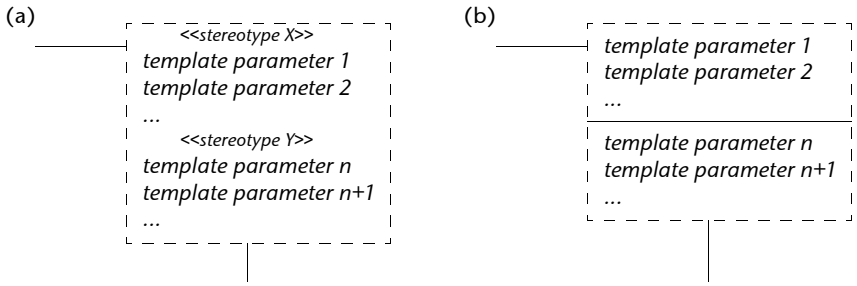


Fig. 9-2 Grouping of template parameters: *(a) template parameters in a stereotyped list, and (b) template parameters in compartments.*

9.2 Diagram Types

AML extends the set of diagram types defined by UML with the following diagram types:

Mental Diagram

A specialized Class Diagram (from UML) used to capture mental attitudes of mental semi-entities in terms of MentalStates, i.e. Goals, Plans, Beliefs and MentalRelationships. Mental Diagrams can be owned either by an MentalSemiEntityType to express its mental model, or by a Package (from UML) to express a shared mental model. For examples see Fig. 6-37 (p. 165) to Fig. 6-42 (p. 169).

Goal-Based Requirements Diagram

A specialized Mental Diagram used to capture goal-based requirements. It usually contains specification of the system stakeholder's mental attitudes concerning the system modeled, and their relationships to the

other elements of the system model. For an example see Fig. 6-23 (p. 155).

Society Diagram

A specialized Class Diagram (from UML) used to capture the global view of the multi-agent system's architecture in terms of EntityTypes (e.g. AgentTypes, ResourceTypes, EnvironmentTypes, OrganizationUnitTypes), EntityRoleTypes, and their relationships (e.g. all kinds of UML relationships, SocialAssociations, PlayAssociations, Perceives and Effects dependencies, ServiceProvisions and ServiceUsages). For examples see Fig. 4-11 (p. 29), Fig. 4-20 (p. 33), Fig. 4-33 (p. 41), Fig. 4-37 (p. 43), and Fig. 4-40 (p. 45).

Entity Diagram

A specialized Composite Structure Diagram (from UML) used to capture the details of the internal structure of an EntityType (in terms of its owned Features, Behaviors and Ports), played EntityRoleTypes, and related ServiceProvisions and ServiceUsages. For examples see Fig. 4-5 (p. 26), Fig. 4-19 (p. 32), Fig. 4-21 (p. 33), Fig. 4-32 (p. 41), Fig. 5-7 (p. 66) part (a), Fig. 5-75 (p. 110), and Fig. 5-98 (p. 121).

Service Diagram

A specialized Composite Structure Diagram (from UML) used to show specification of a service in terms of a ServiceSpecification and owned ServiceInteractionProtocols. For examples see Fig. 5-68 (p. 105), and Fig. 5-73 (p. 108).

Ontology Diagram

A specialized Class Diagram (from UML) used to show a specification of an ontology in terms of Ontologies, OntologyUtilities and OntologyClasses together with their mutual relationships. For examples see Fig. 4-58 (p. 56), Fig. 4-60 (p. 58), and Fig. 4-62 (p. 59).

Behavior Decomposition Diagram

A specialized Class Diagram (from UML) used to show BehavorFragments, owned Capabilities, and their mutual relationships. For an example see Fig. 5-7 (p. 66) part (b).

Protocol Sequence Diagram

A specialized Sequence Diagram (from UML) used to show the specification of an InteractionProtocol in the form of a Sequence Diagram. For an example see Fig. 5-48 (p. 92).

Protocol Communication Diagram

A specialized Communication Diagram (from UML) used to show the specification of an InteractionProtocol in the form of a Communication Diagram. For an example see Fig. 5-49 (p. 93).

Service Protocol Sequence Diagram

A specialized Protocol Sequence Diagram used to show the specification of a ServiceProtocol in the form of Sequence Diagram. For an example see Fig. 5-71 (p. 107).

Service Protocol Communication Diagram

A specialized Protocol Communication Diagram used to show the specification of a ServiceProtocol in the form of Communication Diagram. For an example see Fig. 5-72 (p. 107).

Please note that this taxonomy provides a logical organization for the various major kinds of diagrams. However, it does not preclude the mixing of different kinds of diagram types, as one might do when combining structural and behavioral elements (e.g., showing a state machine nested inside an internal structure). Consequently, the distinction between the various kinds of diagram types are not strictly enforced.

All the specialized Class Diagrams and the Entity Diagram can have two forms: the type and the instance. The instance variations of the diagrams are identified by the word 'Object' in their names, e.g. Mental Object Diagram, Society Object Diagram.

The diagram hierarchy is depicted in Fig. 9-3.

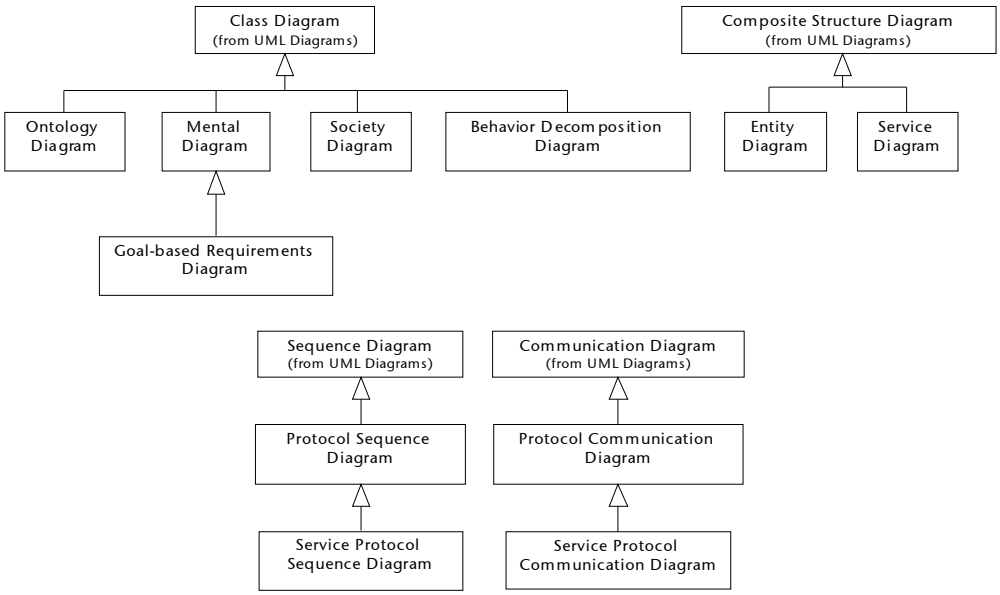


Fig. 9-3 AML diagram taxonomy

10 AML as a UML Profile

This section describes the UML profiles for AML, particularly *UML 2.0 Profile for AML* and *UML 1.* Profile for AML*, see Fig. 1-1 (p. 12).

Each stereotype is specified in a separate subsection, with the following structure:

Stereotype name and description (mandatory)
Title of a subsection specifying the stereotype name. Abstract stereotypes are additionally marked by the keyword '(abstract)', and enumerations are marked by the keyword '(enumeration)'. Text followed by the title gives a description of the stereotype.

Icon (optional)
The stereotype's iconic notation.

Keyword (mandatory for concrete stereotypes)
The stereotype keyword.

Parents (optional)
The stereotypes that are superclasses to the stereotype being defined.

Applies to (mandatory for concrete stereotypes and UML 1.5 stereotypes)
The UML metaclass the stereotype applies to.

Tags (optional)
Definition of the stereotype tags. The tags are defined in terms of their:

- name,
- type,
- multiplicity, and
- natural language semantics explanation.

The structure of the tag definition is as follows:

name: *type*[*multiplicity*]	*Semantics*

Enumeration values (mandatory for enumerations)
The values of the enumeration are described in terms of

- value, and
- semantics.

The default values are marked by the keyword '(default)'.

Constraints (optional)
Set of invariants for the stereotype, which must be satisfied by all instances of that stereotype for the model to be meaningful. The rules thus specify constraints over tagged values and meta-properties defined in the UML metamodel for its UML base metaclass. All constraints are specified in natural language and, where possible, also the corresponding formal OCL [51] specification is given.

10.1 UML 2.0 Profile for AML

Each of the following subsections specifies one stereotype defined in the UML 2.0 profile.

10.1.1 EntityType (abstract)

Represents AML EntityType (p. 23) metaclass.

10.1.2 BehavioralEntityType (abstract)

Represents AML BehavioralEntityType (p. 24) metaclass.

Parents: EntityType (p. 182), BehavioredSemiEntityType (p. 188), and SocializedSemiEntityType (p. 183)

10.1.3 AutonomousEntityType (abstract)

Represents AML AutonomousEntityType (p. 24) metaclass.

Parents: BehavioralEntityType (p. 182) and MentalSemiEntityType (p. 201)

10.1.4 AgentType

Represents AML AgentType (p. 25) metaclass.

Icon: 
Keyword: <<agent>>
Parents: AutonomousEntityType (p. 182)
Applies to: Class

10.1.5 ResourceType

Represents AML ResourceType (p. 27) metaclass.

Icon: 
Keyword: <<resource>>
Parents: BehavioralEntityType (p. 182)
Applies to: Class

10.1.6 EnvironmentType

Represents AML EnvironmentType (p. 28) metaclass.

Icon:
Keyword: <<environment>>
Parents: AutonomousEntityType (p. 182)
Applies to: Class

10.1.7 OrganizationUnitType

Represents AML OrganizationUnitType (p. 31) metaclass.

Icon: 
Keyword: <<organization unit>>
Parents: EnvironmentType (p. 182)
Applies to: Class

10.1.8 SocializedSemiEntityType (abstract)

Represents AML SocializedSemiEntityType (p. 33) metaclass.

Tags:

supportedAcl: String[0..1]	Comma-separated list of identifiers of supported agent communication languages. Represents the AML meta-attribute SocializedSemiEntityType::supportedAcl.
supportedCl: String[0..1]	Comma-separated list of identifiers of supported content languages. Represents the AML meta-attribute SocializedSemiEntityType::supportedCl.
supportedEncoding: String[0..1]	Comma-separated list of identifiers of supported message content encodings. Represents the AML meta-attribute SocializedSemiEntityType::supportedEncoding.
supportedOntology: String[0..1]	Comma-separated list of identifiers of supported ontologies. Represents the AML meta-attribute SocializedSemiEntityType::supportedOntology.

10.1.9 SocialProperty (abstract)

Represents AML SocialProperty (p. 35) metaclass. Its concrete subclasses represent SocialProperties of different social role kinds.

Parents: ServicedProperty (p. 195)
Applies to: Property
Constraints:

1. SocialProperty can be owned only by a SocializedSemiEntityType (p. 183) or a SocialAssociation (p. 184):

   ((self.class->notEmpty() implies
   self.class.oclIsKindOf(SocializedSemiEntityType)) or
   (self.owningAssociation->notEmpty() implies
   self.owningAssociation.oclIsKindOf(SocialAssociation)))

2. If the type meta-association is specified, the element referred to by it must be a SocializedSemiEntityType:

   self.type->notEmpty() implies
   self.type.oclIsKindOf(SocializedSemiEntityType)

10.1.10 SocialPropertyPeer

Represents AML SocialProperty (p. 35) metaclass with the socialRole meta-attribute set to *peer*.

Icon: ◮
Keyword: <<peer>>
Parents: SocialProperty (p. 183)
Applies to: Property

10.1.11 SocialPropertySuperordinate

Represents AML SocialProperty (p. 35) metaclass with the socialRole meta-attribute set to *superordinate*.

Icon: ▲
Keyword: <<super>>
Parents: SocialProperty (p. 183)
Applies to: Property

10.1.12 SocialPropertySubordinate

Represents AML SocialProperty (p. 35) metaclass with the socialRole meta-attribute set to *subordinate*.

Icon: △
Keyword: <<sub>>
Parents: SocialProperty (p. 183)
Applies to: Property

10.1.13 SocialAssociation

Represents AML SocialAssociation (p. 38) metaclass.

Keyword: <<social>>
Applies to: Association
Constraints:

1. The memberEnd meta-association refers only to SocialProperties (p. 183):

   self.memberEnd->forAll(oclIsKindOf(SocialProperty))

10.1.14 EntityRoleType

Represents AML EntityRoleType (p. 39) metaclass.

Icon:
Keyword: <<entity role>>
Parents: BehavioredSemiEntityType (p. 188), MentalSemiEntityType (p. 201), and SocializedSemiEntityType (p. 183)
Applies to: Class

10.1.15 RoleProperty

Represents AML RoleProperty (p. 41) metaclass.

Keyword: <<role>>
Applies to: Property
Constraints:

1. RoleProperty can be owned only by a BehavioralEntityType (p. 182) or a PlayAssociation (p. 185):

   ((self.class->notEmpty() implies
   self.class.oclIsKindOf(BehavioralEntityType)) or
   (self.owningAssociation->notEmpty() implies
   self.owningAssociation.oclIsKindOf(PlayAssociation)))

2. If the type meta-association is specified, the element referred to by it must be an EntityRoleType (p. 184):

   self.type->notEmpty() implies
   self.type.oclIsKindOf(EntityRoleType)

10.1.16 PlayAssociation

Represents AML PlayAssociation (p. 43) metaclass.

Keyword: <<play>>
Applies to: Association
Constraints:

1. PlayAssociation is binary:

   self.memberEnd->size()=2

2. One memberEnd must refer to a RoleProperty (p. 184):

   self.memberEnd->exists(me|me.oclIsKindOf(RoleProperty))

3. If the multiplicity of the other memberEnd is specified, it must be at most 1:

   self.memberEnd->exists(me| not me.oclIsKindOf(RoleProperty) and
   me.upperBound()<=1)

10.1.17 CreateRoleAction

Represents AML CreateRoleAction (p. 45) metaclass. One of its OutputPins represents the created entity role.

Icon: 
Keyword: <<create role>>
Applies to: Action

Tags:

player: String[1]	Specification of the player of the created entity role. Represents the AML meta-association CreateRoleAction::player.
roleProperty: String[1]	Specification of the RoleProperty (p. 184) where the created entity role is being placed. Represents the AML meta-association CreateRoleAction::roleProperty.
roleType: String[1]	Specification of the instantiated EntityRoleType (p. 184). Represents the AML meta-association CreateRoleAction::roleType.

Constraints:

1. The player tagged value must refer to an existing behavioral entity (e.g. by its name or an expression evaluated to a behavioral entity). This constraint cannot be expressed in OCL.
2. The roleProperty tagged value must be the name of a RoleProperty owned by the player's type/classifier. This constraint cannot be expressed in OCL.
3. The roleType tagged value must be the name of an existing EntityRoleType. This constraint cannot be expressed in OCL.
4. The type of the OutputPin representing the created entity role must conform to the type specified by the roleType tagged value. This constraint cannot be expressed in OCL.
5. The EntityRoleType referred to by the roleType tagged value must conform to the type of the RoleProperty referred to by the roleProperty tagged value. This constraint cannot be expressed in OCL.

10.1.18 DisposeRoleAction

Represents AML DisposeRoleAction (p. 47) metaclass. It destroys the entity roles represented by its InputPins.

Icon: 
Keyword: <<dispose role>>
Applies to: DestroyObjectAction

10.1.19 AgentExecutionEnvironment

Represents AML AgentExecutionEnvironment (p. 49) metaclass.

Icon: 
Keyword: <<agent execution environment>>
Parents: BehavioredSemiEntityType (p. 188)
Applies to: ExecutionEnvironment
Constraints:

1. The internal structure of an AgentExecutionEnvironment (p. 186) can also consist of other attributes than parts of the type Node (from UML).

The constraint [1] defined for UML Node, and inherited by the AgentExecutionEnvironment, is therefore discarded (see [53] for details).

10.1.20 HostingProperty

Represents AML HostingProperty (p. 51) metaclass.

Icon:  for hosting of an agent,
 for hosting of a resource,
 for hosting of an environment,
 for hosting of an organization unit
Keyword: <<hosting>>
Parents: ServicedProperty (p. 195)
Applies to: Property
Tags:

hostedAs: String[0..1]	The set of hosting kinds that the owning AgentExecutionEnvironment (p. 186) provides to the HostingProperty's type. The set is specified as a comma-separated list of keywords defined by the HostingKind (p. 54) enumeration from the AML metamodel. Represents the AML meta-attribute HostingProperty::hostingKind.

Constraints:

1. HostingProperty can be owned only by an AgentExecutionEnvironment or a HostingAssociation (p. 187):

   ((self.class->notEmpty() implies
   self.class.oclIsKindOf(AgentExecutionEnvironment)) or
   (self.owningAssociation->notEmpty() implies
   self.owningAssociation.oclIsKindOf(HostingAssociation)))

2. If the type meta-association is specified, the element referred to by it must be an EntityType (p. 182):

   self.type->notEmpty() implies
   self.type.oclIsKindOf(EntityType)

10.1.21 HostingAssociation

Represents AML HostingAssociation (p. 54) metaclass.

Keyword: <<hosting>>
Applies to: Association
Constraints:

1. HostingAssociation is binary:

   self.memberEnd->size()=2

2. One memberEnd must refer to a HostingProperty (p. 187):

   self.memberEnd->exists(me|me.oclIsKindOf(HostingProperty))

10.1.22 Ontology

Represents AML Ontology (p. 56) metaclass.

Icon: 
Keyword: <<ontology>>
Applies to: Package

10.1.23 OntologyClass

Represents AML OntologyClass (p. 57) metaclass.

Icon: 
Keyword: <<oclass>>
Applies to: Class

10.1.24 OntologyUtility

Represents AML OntologyUtility (p. 58) metaclass.

Icon: 
Keyword: <<outility>>
Applies to: Class

10.1.25 BehavioredSemiEntityType (abstract)

Represents AML BehavioredSemiEntityType (p. 60) metaclass.

Parents: ServicedElement (p. 195)

10.1.26 BehaviorFragment

Represents AML BehaviorFragment (p. 65) metaclass.

Icon: 
Keyword: <<behavior fragment>>
Parents: BehavioredSemiEntityType (p. 188)
Applies to: Class

10.1.27 MultiLifeline

Represents AML MultiLifeline (p. 70) metaclass.

Keyword: <<multi>> (but is usually omitted from diagrams)
Applies to: Lifeline
Tags:

multiplicity: String[1]	The multiplicity string as defined by UML. It is specified without square brackets.

10.1.28 MultiMessage

Represents AML MultiMessage (p. 72) metaclass.

Icon: 
Keyword: <<multi>>

Applies to: Message
Tags:

sendDiscriminator: String[0..1]	The constraint which specifies the MultiMessage senders when it is sent from a MultiLifeline (p. 188). Senders are those instances represented by the MultiLifeline for which the sendDiscriminator is evaluated to *true*. Represents the AML meta-association MultiMessage::sendDiscriminator.
receiveDiscriminator: String[0..1]	The constraint which specifies the MultiMessage receivers when it is sent to a MultiLifeline. Receivers are those instances represented by the MultiLifeline for which the receiveDiscriminator is evaluated to *true*. Represents the AML meta-association MultiMessage::receiveDiscriminator.
toItself: Boolean[1]	If *true*, the MultiMessage is sent also to its sender when the sender belongs to the group of receivers. If *false*, the sender is excluded from the group of receivers. Represents the AML meta-attribute MultiMessage::toItself.

Constraints:

1. At least one end of the MultiMessage must be a MultiLifeline:

   self.sendEvent.covered.oclIsKindOf(MultiLifeline) or
   self.receiveEvent.covered.oclIsKindOf(MultiLifeline)

2. The sendDiscriminator tagged value can be specified only if the sender is represented by a MultiLifeline. This constraint cannot be expressed in OCL.

3. The receiveDiscriminator tagged value can be specified only if the receiver is represented by a MultiLifeline. This constraint cannot be expressed in OCL.

10.1.29 DecoupledMessage

Represents AML DecoupledMessage (p. 74) metaclass.

Icon: ⚡
Keyword: <<decoupled>>
Parents: MultiMessage (p. 188)
Applies to: Message
Tags:

payload: String[0..1]	The type of the object transmitted, specified as the name of a DecoupledMessagePayload (p. 190). Represents the AML meta-association DecoupledMessage::payload.

Constraints:

1. The payload tagged value must be the name of an existing DecoupledMessagePayload. This constraint cannot be expressed in OCL.

2. The constraints [2], [3], and [4] imposed on the UML Message are released, i.e. the DecoupledMessage's (p. 195) signature does not need to refer to either an Operation or a Signal (both from UML).

10.1.30 DecoupledMessagePayload

Represents AML DecoupledMessagePayload (p. 76) metaclass.

Icon: 
Keyword: <<dm payload>>
Applies to: Class

10.1.31 Subset

Represents AML Subset (p. 76) metaclass.

Keyword: <<sub>>
Applies to: Dependency
Constraints:

1. Subset has one client:

   self.client->size()=1

2. The client of a Subset is a UML EventOccurrence and the suppliers of a Subset are UML Lifelines:

   self.client->forAll(oclIsKindOf(EventOccurrence)) and
   self.supplier->forAll(oclIsKindOf(Lifeline))

3. All types of the supplier Lifelines must conform to the type of the client's Lifeline:

   self.supplier.oclAsType(Lifeline)->forAll(
   (represents.type->notEmpty() and
   self.client.oclAsType(EventOccurrence).covered.represents.type
   ->notEmpty()) implies
   self.supplier.oclAsType(Lifeline).represents.type.conformsTo(
   self.client.oclAsType(EventOccurrence).covered.represents.type))

10.1.32 Join

Represents AML Join (p. 78) metaclass.

Keyword: <<join>>
Applies to: Dependency
Tags:

selector: String[0..1]	Specifies the set of instances being joined. Represents the AML meta-association Join::selector.

Constraints:

1. Join has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client and supplier of a Join are UML EventOccurrences:

self.client.oclIsKindOf(EventOccurrence) and
self.supplier.oclIsKindOf(EventOccurrence)

3. The Lifeline owning the supplier must be a MultiLifeline (p. 188):

   self.supplier.oclAsType(EventOccurrence).covered.
   oclIsKindOf(MultiLifeline)

4. The type of the client's Lifeline must conform to the type of the supplier's MultiLifeline:

   (self.client.oclAsType(EventOccurrence).covered.represents.type
   ->notEmpty() and
   self.supplier.oclAsType(EventOccurrence).covered.represents.type
   ->notEmpty()) implies
   self.client.oclAsType(EventOccurrence).covered.represents.type.
   conformsTo(self.supplier.oclAsType(EventOccurrence).covered.
   represents.type)

10.1.33 CreateAttribute

Represents AML AttributeChange's (p. 81) createdLifeline meta-association.

Keyword: <<create attribute>>
Applies to: Dependency
Constraints:

1. CreateAttribute has one client:

   self.client->size()=1

2. The client of a CreateAttribute is an EventOccurrence and the suppliers are Lifelines:

   self.client.oclIsKindOf(EventOccurrence) and
   self.supplier->forAll(oclIsKindOf(Lifeline))

3. Each supplier (a Lifeline) must represent an attribute of a Classifier used as the type of the ConnectableElement represented by the Lifeline covering the CreateAttribute's client (an EventOccurrence):

   self.client.oclAsType(EventOccurrence).covered.represents.type
   ->notEmpty() implies
   self.client.oclAsType(EventOccurrence).covered.represents.type.
   attribute->includesAll(self.supplier.represents)

10.1.34 DestroyAttribute

Represents AML AttributeChange's (p. 81) destroyedLifeline meta-association.

Keyword: <<destroy attribute>>
Applies to: Dependency
Constraints:

1. DestroyAttribute has one client:

   self.client->size()=1

2. The client and suppliers of a DestroyAttribute are EventOccurrences:

   self.client.oclIsKindOf(EventOccurrence) and
   self.supplier.oclIsKindOf(EventOccurrence)

3. Each supplier (an EventOccurrence) must represent an attribute of the Classifier used as the type of the ConnectableElement represented by the Lifeline covering the DestroyAttribute's client (an EventOccurrence):

   self.client.oclAsType(EventOccurrence).covered.represents.type
   ->notEmpty() implies
   self.client.oclAsType(EventOccurrence).covered.represents.type.
   attribute->includesAll(self.supplier.covered.represents)

10.1.35 CommunicationSpecifier (abstract)

Represents AML CommunicationSpecifier (p. 85) metaclass.

Tags:

acl: String[0..1]	Denotes the agent communication language in which CommunicationMessages (p. 192) are expressed. Represents the AML meta-attribute CommunicationSpecifier::acl.
cl: String[0..1]	Denotes the language in which the CommunicationMessage's content is expressed, also called the content language. Represents the AML meta-attribute CommunicationSpecifier::cl.
encoding: String[0..1]	Denotes the specific encoding of the CommunicationMessage's content. Represents the AML meta-attribute CommunicationSpecifier::encoding.
ontology: String[0..1]	Comma-separated list of identifiers of ontologies used to give a meaning to the symbols in the CommunicationMessage's content expression. Represents the AML meta-attribute CommunicationSpecifier::ontology.

10.1.36 CommunicationMessage

Represents AML CommunicationMessage (p. 86) metaclass.

Icon:
Keyword: <<communication>>
Parents: DecoupledMessage (p. 189) and CommunicationSpecifier (p. 192)
Applies to: Message
Tags:

payload: String[0..1]	The type of the object transmitted, specified as the name of a CommunicationMessagePayload (p. 193). Represents the AML meta-association CommunicationMessage::payload.

Constraints:

1. The payload tagged value must be the name of an existing CommunicationMessagePayload. This constraint cannot be expressed in OCL.

10.1.37 CommunicationMessagePayload

Represents AML CommunicationMessagePayload (p. 87) metaclass.

Icon: 
Keyword: <<cm payload>>
Parents: DecoupledMessagePayload (p. 190)
Applies to: Class
Tags:

performative: String[0..1]	Performative of the CommunicationMessagePayload. Represents the AML meta-attribute CommunicationMessagePayload::performative.

10.1.38 CommunicativeInteraction

Represents AML CommunicativeInteraction (p. 89) metaclass. If the CommunicativeInteraction is parametrized, it represents the InteractionProtocol (p. 89) from the AML metamodel.

Keyword: <<communicative>> (but is usually omitted from diagrams)
Parents: CommunicationSpecifier (p. 192)
Applies to: Interaction

10.1.39 SendDecoupledMessageAction

Represents AML SendDecoupledMessageAction (p. 94) metaclass.

Icon: 
Keyword: <<send decoupled message>>
Applies to: SendObjectAction

10.1.40 SendCommunicationMessageAction

Represents AML SendCommunicationMessageAction (p. 95) metaclass.

Icon: 
Keyword: <<send communication message>>
Parents: SendDecoupledMessageAction (p. 193) and CommunicationSpecifier (p. 192)
Applies to: SendObjectAction

10.1.41 AcceptDecoupledMessageAction

Represents AML AcceptDecoupledMessageAction (p. 97) metaclass.

Icon: 
Keyword: <<accept decoupled message>>
Applies to: AcceptEventAction
Constraints:

1. If the type of the OutputPin referred to by the result meta-association is specified, it must be a DecoupledMessagePayload (p. 190):

   self.result.type->notEmpty() implies
   self.result.type.oclIsKindOf(DecoupledMessagePayload)

10.1.42 AcceptCommunicationMessageAction

Represents AML AcceptCommunicationMessageAction (p. 98) metaclass.

Icon: 
Keyword: <<accept communication message>>
Applies to: AcceptEventAction
Constraints:

1. If the type of the OutputPin referred to by the result meta-association is specified, it must be a CommunicationMessagePayload (p. 193):

   self.result.type->notEmpty() implies
   self.result.type.oclIsKindOf(CommunicationMessagePayload)

10.1.43 ServiceSpecification

Represents AML ServiceSpecification (p. 103) metaclass.

Icon: 
Keyword: <<service specification>>
Parents: CommunicationSpecifier (p. 192)
Applies to: Class

10.1.44 ProviderParameter

Represents a special TemplateParameter (from UML) used to represent the type of the AML ServiceProtocol::providerParameter meta-association. ServiceProtocol (p. 105) from the AML metamodel is represented in the UML profile as a parametrized CommunicativeInteraction (p. 193) stereotype with its TemplateParameters stereotyped either by the ProviderParameter or the ClientParameter (p. 194) stereotypes.

Keyword: <<provider>>
Applies to: TemplateParameter
Constraints:

1. ProviderParameter can be used only in the TemplateSignature (from UML) of a CommunicativeInteraction:

   self.signature.template.oclIsKindOf(CommunicativeInteraction)

10.1.45 ClientParameter

Represents a special TemplateParameter (from UML) used to represent the type of the AML ServiceProtocol::clientParameter meta-association. ServiceProtocol (p. 105) from the AML metamodel is represented in the UML profile as a parametrized CommunicativeInteraction (p. 193) with its Tem-

plateParameters stereotyped either by the ProviderParameter (p. 194) or the ClientParameter stereotypes.

Keyword: <<client>>
Applies to: TemplateParameter
Constraints:

1. ClientParameter can be used only in the TemplateSignature (from UML) of a CommunicativeInteraction:

   self.signature.template.oclIsKindOf(CommunicativeInteraction)

10.1.46 ServicedElement (abstract)

Represents AML ServicedElement (p. 108) metaclass.

10.1.47 ServicedProperty

Represents AML ServicedProperty (p. 109) metaclass.

Keyword: <<serviced>>
Parents: ServicedElement (p. 195)
Applies to: Property
Constraints:

1. If the type meta-association is specified, the element referred to by it must be a BehavioredSemiEntityType (p. 188):

   self.type->notEmpty() implies
   self.type.oclIsKindOf(BehavioredSemiEntityType)

10.1.48 ServicedPort

Represents AML ServicedPort (p. 110) metaclass.

Keyword: <<serviced>>
Parents: ServicedElement (p. 195)
Applies to: Port
Constraints:

1. If the type meta-association is specified, the element referred to by it must be a BehavioredSemiEntityType (p. 188):

   self.type->notEmpty() implies
   self.type.oclIsKindOf(BehavioredSemiEntityType)

10.1.49 ServiceProvision

Represents AML ServiceProvision (p. 112) metaclass. Provider template parameter substitutions are specified as the body of the attached UML Comment.

Keyword: <<provides>>
Applies to: Realization
Constraints:

1. ServiceProvision has one client and one supplier:

self.client->size()=1 and self.supplier->size()=1

2. The client of a ServiceProvision is a ServicedElement (p. 195) and the supplier of a ServiceProvision is a ServiceSpecification (p. 194):

   self.client.oclIsKindOf(ServicedElement) and
   self.supplier.oclIsKindOf(ServiceSpecification)

3. The specified provider template parameter substitutions bind all (and only) the provider parameters from all service protocols of the ServiceSpecification. This constraint cannot be expressed in OCL.

10.1.50 ServiceUsage

Represents AML ServiceUsage (p. 114) metaclass. Client template parameter substitutions are specified as the body of the attached UML Comment.

Keyword: <<uses>>
Applies to: Usage
Constraints:

1. ServiceUsage has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client of a ServiceUsage is a ServicedElement (p. 195) and the supplier of a ServiceUsage is a ServiceSpecification (p. 194):

   self.client.oclIsKindOf(ServicedElement) and
   self.supplier.oclIsKindOf(ServiceSpecification)

3. The specified client template parameter substitutions bind all (and only) the client parameters from all service protocols of the ServiceSpecification. This constraint cannot be expressed in OCL.

10.1.51 PerceivingAct

Represents AML PerceivingAct (p. 117) metaclass.

Keyword: <<pa>>
Applies to: Operation
Constraints:

1. PerceivingAct can be owned only by a PerceptorType (p. 196):

   self.class.oclIsKindOf(PerceptorType)

10.1.52 PerceptorType

Represents AML PerceptorType (p. 118) metaclass.

Icon: 
Keyword: <<perceptor type>>
Parents: BehavioredSemiEntityType (p. 188)
Applies to: Class

10.1.53 Perceptor

Represents AML Perceptor (p. 119) metaclass.

Icon: 
Keyword: <<perceptor>>
Parents: ServicedPort (p. 195)
Applies to: Port
Constraints:

1. Perceptor can be owned only by a BehavioredSemiEntityType (p. 188):

   self.redefinitionContext.oclIsKindOf(BehavioredSemiEntityType)

2. If the type meta-association is specified, the element referred to by it must be a PerceptorType (p. 196):

   self.type->notEmpty() implies self.type.oclIsKindOf(PerceptorType)

10.1.54 PerceptAction

Represents AML PerceptAction (p. 121) metaclass.

Icon: 
Keyword: <<percept>>
Applies to: CallOperationAction
Constraints:

1. The operation meta-association must refer to a PerceivingAct (p. 196):

   self.operation.oclIsKindOf(PerceivingAct)

10.1.55 Perceives

Represents AML Perceives (p. 123) metaclass.

Keyword: <<perceives>>
Applies to: Dependency

10.1.56 EffectingAct

Represents AML EffectingAct (p. 123) metaclass.

Keyword: <<ea>>
Applies to: Operation
Constraints:

1. EffectingAct can be owned only by an EffectorType (p. 197):

   self.class.oclIsKindOf(EffectorType)

10.1.57 EffectorType

Represents AML EffectorType (p. 124) metaclass.

Icon: 
Keyword: <<effector type>>

Parents: BehavioredSemiEntityType (p. 188)
Applies to: Class

10.1.58 Effector

Represents AML Effector (p. 125) metaclass.

Icon: 
Keyword: <<effector>>
Parents: ServicedPort (p. 195)
Applies to: Port
Constraints:

1. Effector can be owned only by a BehavioredSemiEntityType (p. 188):

   self.redefinitionContext.oclIsKindOf(BehavioredSemiEntityType)

2. If the type meta-association is specified, the element referred to by it must be an EffectorType (p. 197):

   self.type->notEmpty() implies self.type.oclIsKindOf(EffectorType)

10.1.59 EffectAction

Represents AML EffectAction (p. 126) metaclass.

Icon: 
Keyword: <<effect>>
Applies to: CallOperationAction
Constraints:

1. The operation meta-association must refer to an EffectingAct (p. 197):

   self.operation.oclIsKindOf(EffectingAct)

10.1.60 Effects

Represents AML Effects (p. 127) metaclass.

Keyword: <<effects>>
Applies to: Dependency

10.1.61 Move

Represents AML Move (p. 129) metaclass.

Keyword: <<move>>
Applies to: Dependency
Constraints:

1. Move has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client and supplier of a Move are HostingProperties (p. 187):

   self.client.oclIsKindOf(HostingProperty) and
   self.supplier.oclIsKindOf(HostingProperty)

3. The type of the HostingProperty at the supplier end must conform to the type of the HostingProperty at the client end, if both are specified:

   (self.client.oclAsType(HostingProperty).type->notEmpty() and
   self.supplier.oclAsType(HostingProperty).type->notEmpty()) implies
   self.supplier.oclAsType(HostingProperty).type.
   conformsTo(self.client.oclAsType(HostingProperty).type)

10.1.62 Clone

Represents AML Clone (p. 130) metaclass.

Keyword: <<clone>>
Applies to: Dependency
Constraints:

1. Clone has one client:

   self.client->size()=1

2. The client and suppliers of a Clone are HostingProperties (p. 187):

   self.client.oclIsKindOf(HostingProperty) and
   self.supplier->forAll(oclIsKindOf(HostingProperty))

3. If specified, the types of the HostingProperties at the supplier ends must conform to the type of the HostingProperty at the client end:

   (self.client.oclAsType(HostingProperty).type->notEmpty() and
   self.supplier.oclAsType(HostingProperty).type->notEmpty()) implies
   self.supplier.oclAsType(HostingProperty).type->
   forAll(conformsTo(self.client.oclAsType(HostingProperty).type))

10.1.63 MobilityAction (abstract)

Represents AML MobilityAction (p. 131) metaclass.

Applies to: AddStructuralFeatureValueAction
Constraints:

1. If the type of the InputPin referred to by the value meta-association is specified, it must be an EntityType (p. 182):

   self.value.type->notEmpty() implies
   self.value.type.oclIsKindOf(EntityType)

2. If the type of the InputPin referred to by the object meta-association is specified, it must be an AgentExecutionEnvironment (p. 186):

   self.object.type->notEmpty() implies
   self.object.type.oclIsKindOf(AgentExecutionEnvironment)

3. The structuralFeature meta-association must refer to a HostingProperty (p. 187):

   self.structuralFeature.oclIsKindOf(HostingProperty)

4. If the type of the InputPin referred to by the object meta-association is specified, the HostingProperty referred to by the structuralFeature meta-association must be an owned attribute of that type:

   self.object.type->notEmpty() implies
   self.object.type.ownedAttribute->includes(self.structuralFeature)

10.1.64 MoveAction

Represents AML MoveAction (p. 132) metaclass.

Icon: 
Keyword: <<move>>
Parents: MobilityAction (p. 199)
Applies to: AddStructuralFeatureValueAction

10.1.65 CloneAction

Represents AML CloneAction (p. 133) metaclass. One of its OutputPins represents the entity clone.

Icon: 
Keyword: <<clone>>
Parents: MobilityAction (p. 199)
Applies to: AddStructuralFeatureValueAction

10.1.66 MentalState (abstract)

Represents AML MentalState (p. 139) metaclass.

Tags:

degree: String[0..1]	The degree of a MentalState. Represents the AML meta-attribute MentalState::degree.

10.1.67 MentalClass (abstract)

Represents AML MentalClass (p. 140) metaclass.

Parents: MentalState (p. 200)

10.1.68 ConstrainedMentalElement (abstract)

This stereotype is used to define the possibility of specifying MentalConstraints (p. 141) from the AML metamodel as tagged values, but does not directly represent any AML metaclass. It is used as a common superclass to the ConstrainedMentalClass (p. 201) and MentalProperty (p. 201) stereotypes.

Tags:

commitCondition: String[0..1]	An assertion identifying the situation under which an autonomous entity commits to the ConstrainedMentalClass (if the precondition also holds). Represents the AML meta-attribute MentalConstraintKind::commitCondition.

preCondition: String[0..1]	The condition that must hold before the ConstrainedMentalClass can become effective (i.e. a goal can be committed to or a plan can be executed). Represents the AML meta-attribute MentalConstraintKind::preCondition.
invariant: String[0..1]	The condition that holds during the period the ConstrainedMentalClass remains effective. Represents the AML meta-attribute MentalConstraintKind::invariant.
cancelCondition: String[0..1]	An assertion identifying the situation under which an autonomous entity cancels attempting to accomplish the ConstrainedMentalClass. Represents the AML meta-attribute MentalConstraintKind::cancelCondition.
postCondition: String[0..1]	The condition that holds after the ConstrainedMentalClass has been accomplished (i.e. a goal has been achieved or a plan has been executed). Represents the AML meta-attribute MentalConstraintKind::postCondition.

10.1.69 ConstrainedMentalClass (abstract)

Represents AML ConstrainedMentalClass (p. 140) metaclass.

Parents: MentalClass (p. 200) and ConstrainedMentalElement (p. 200)

10.1.70 MentalRelationship (abstract)

Represents AML MentalRelationship (p. 143) metaclass.

Parents: MentalState (p. 200)

10.1.71 MentalSemiEntityType (abstract)

Represents AML MentalSemiEntityType (p. 143) metaclass.

10.1.72 MentalProperty

Represents AML MentalProperty (p. 144) metaclass.

Keyword: <<mental>>
Parents: ConstrainedMentalElement (p. 200)
Applies to: Property
Tags:

degree: String[0..1]	The degree of the MentalClass (p. 200) specified as the type of the MentalProperty. If specified, it overrides the degree tagged value specified for the MentalClass itself. Represents the AML meta-attribute MentalProperty::degree.

Constraints:

1. MentalProperty can be owned only by a MentalSemiEntityType (p. 201) or a MentalAssociation (p. 202):

   ((self.class->notEmpty() implies
   self.class.oclIsKindOf(MentalSemiEntityType)) or
   (self.owningAssociation->notEmpty() implies
   self.owningAssociation.oclIsKindOf(MentalAssociation)))

2. If the type meta-association is specified, the element referred to by it must be a MentalClass (p. 200):

   self.type->notEmpty() implies
   self.type.oclIsKindOf(MentalClass)

3. If the type meta-association is specified, the MentalClass referred to by it cannot be a Plan (p. 203):

   self.type->notEmpty() implies (not self.type.oclIsKindOf(Plan))

10.1.73 MentalAssociation

Represents AML MentalAssociation (p. 147) metaclass.

Keyword: <<mental>>
Applies to: Association
Constraints:

1. MentalAssociation is binary:

   self.memberEnd->size()=2

2. One memberEnd must refer to a MentalProperty (p. 201):

   self.memberEnd->exists(me|me.oclIsKindOf(MentalProperty))

10.1.74 Responsibility

Represents AML Responsibility (p. 148) metaclass.

Keyword: <<responsible>>
Applies to: Realization
Constraints:

1. The suppliers of a Responsibility are MentalClasses (p. 200):

   self.supplier->forAll(oclIsKindOf(MentalClass))

10.1.75 Belief

Represents AML Belief (p. 149) metaclass.

Icon: 
Keyword: <<belief>>
Parents: MentalClass (p. 200)

Applies to: Class
Tags:

constraint: String[0..1]	The specification of the information a Belief represents. Represents the AML meta-attribute Belief::constraint.

10.1.76 Goal (abstract)

Represents AML Goal (p. 151) metaclass.

Parents: ConstrainedMentalClass (p. 201)

10.1.77 DecidableGoal

Represents AML DecidableGoal (p. 152) metaclass.

Icon: 
Keyword: <<dgoal>>
Parents: Goal (p. 203)
Applies to: Class

10.1.78 UndecidableGoal

Represents AML UndecidableGoal (p. 154) metaclass.

Icon: 
Keyword: <<ugoal>>
Parents: Goal (p. 203)
Applies to: Class

10.1.79 Plan

Represents AML Plan (p. 156) metaclass.

Icon: 
Keyword: <<plan>>
Parents: ConstrainedMentalClass (p. 201)
Applies to: Activity
Constraints:

1. If the context (see UML Behavior::context meta-association) for a Plan is specified, it must be a MentalSemiEntityType (p. 201):

   self.context->notEmpty() implies
   self.context.oclIsKindOf(MentalSemiEntityType)

10.1.80 CommitGoalAction

Represents AML CommitGoalAction (p. 158) metaclass. One of its OutputPins represents the created Goal (p. 151) instance.

Icon: 
Keyword: <<commit goal>>
Applies to: Action

Tags:

mentalSemiEntity: String[1]	Specification of the mental semi-entity committed to the Goal (p. 203). Represents the AML meta-association CommitGoalAction::mentalSemiEntity.
mentalProperty: String[1]	The name of the MentalProperty (p. 201) where the created Goal instance is being placed. Represents the AML meta-association CommitGoalAction::mentalProperty.
goalType: String[1]	The name of the instantiated Goal. Represents the AML meta-association CommitGoalAction::goalType.

Constraints:

1. The mentalSemiEntity tagged value must refer to an existing mental semi-entity (e.g. by its name or an expression evaluated to a mental semi-entity). This constraint cannot be expressed in OCL.
2. The mentalProperty tagged value must be the name of a MentalProperty owned by the mentalSemiEntity's type. This constraint cannot be expressed in OCL.
3. The goalType tagged value must be the name of an existing Goal. This constraint cannot be expressed in OCL.
4. If the type of the OutputPin representing the created Goal instance is specified, it must conform to the type specified by the goalType tagged value. This constraint cannot be expressed in OCL.
5. The Goal referred to by the goalType tagged value must conform to the type of the MentalProperty referred to by the mentalProperty tagged value. This constraint cannot be expressed in OCL.

10.1.81 CancelGoalAction

Represents AML CancelGoalAction (p. 160) metaclass. It destroys the Goal (p. 151) instances represented by its InputPins.

Icon: 
Keyword: <<cancel goal>>
Applies to: DestroyObjectAction

10.1.82 Contribution

Represents AML Contribution (p. 162) metaclass. It is represented by a binary UML Dependency relationship. A Contribution's contributor is represented by the Dependency client and the beneficiary by the Dependency supplier.

Alternatively, the Contribution relationship from the AML metamodel can be represented by UML Association with a stereotype representing its kind (<<sufficient>>, <<necessary>>, or <<iff>>; see the ContributionKind, p. 169 enumeration for details), and its tagged values contributorConstraintKind and beneficiaryConstraintKind represented as association role properties

(with the keywords: *commit*, *pre*, *commpre*, *inv*, *cancel*, or *post*; see the MentalConstraintKind, p. 142 enumeration for details).

Keyword: <<contributes>>
Parents: MentalRelationship (p. 201)
Applies to: Dependency
Tags:

kind: ContributionKind[1]	Determines whether the contribution of the client's tagged value representing a MentalConstraint (p. 141) (from the AML Metamodel) of a specified kind is a necessary, sufficient, or equivalent condition for the supplier's tagged value representing a MentalConstraint (from the AML Metamodel) of a specified kind. Represents the AML meta-association Contribution::kind.
contributorConstraintKind: MentalConstraintKind [0..1]	The kind of the client's tagged value representing a MentalConstraint (from the AML Metamodel) which contributes to the kind of the supplier's tagged value representing a MentalConstraint (from the AML Metamodel). Represents the AML meta-association Contribution::contributorConstraintKind.
beneficiaryConstraintKind: MentalConstraintKind [0..1]	The kind of the supplier's tagged value representing a MentalConstraint (from the AML Metamodel) to which the kind of the client's tagged value representing a MentalConstraint (from the AML Metamodel) contributes. Represents the AML meta-association Contribution::beneficiaryConstraintKind.

Constraints:

1. Contribution has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client and supplier of a Contribution are MentalStates (p. 200):

   self.client.oclIsKindOf(MentalState) and
   self.supplier.oclIsKindOf(MentalState)

3. If the MentalState referred to by the client meta-association is a Belief (p. 202) or a Contribution, the contributorConstraintKind tagged value is unspecified:

   (self.client.oclIsKindOf(Belief) or
   self.client.oclIsKindOf(Contribution)) implies
   self.contributorConstraintKind=#unspecified

4. If the MentalState referred to by the supplier meta-association is a Belief or a Contribution, the beneficiaryConstraintKind tagged value is unspecified:

   (self.supplier.oclIsKindOf(Belief) or
   self.supplier.oclIsKindOf(Contribution)) implies
   self.beneficiaryConstraintKind=#unspecified

10.1.83 ContributionKind (enumeration)

Represents AML ContributionKind (p. 169) metaclass.

Enumeration values:

sufficient (default)	The contributor or its tagged value representing a MentalConstraint (p. 141) (from the AML Metamodel) of the given kind (if specified) is a sufficient condition for the beneficiary or its tagged value representing a MentalConstraint (from the AML Metamodel) of the given kind (if specified). Represents the enumeration value ContributionKind::sufficient from the AML metamodel.
necessary	The contributor or its tagged value representing a MentalConstraint (from the AML Metamodel) of the given kind (if specified) is a necessary condition for the beneficiary or its tagged value representing a MentalConstraint (from the AML Metamodel) of the given kind (if specified). Represents the enumeration value ContributionKind::necessary from the AML metamodel.
iff	The contributor and beneficiary or their tagged value representing MentalConstraints (from the AML Metamodel) of the given kinds (if specified) are equivalent. Represents the enumeration value ContributionKind::iff from the AML metamodel.

10.1.84 MentalConstraintKind (enumeration)

Represents AML MentalConstraintKind (p. 142) metaclass. Its values indicate usage of the MentalConstraints (p. 141) from the AML metamodel specified as tagged values of elements stereotyped by any concrete ConstrainedMentalElement (p. 200) in a relationship stereotyped by Contribution (p. 204).

Enumeration values:

unspecified (default)	Mental constraint kind is unspecified.
commit	Indicates the *commitCondition*.
pre	Indicates the *preCondition*.
commpre	Indicates an AND-ed combination of *commitCondition* and *preCondition*.
inv	Indicates the *invariant*.
cancel	Indicates the *cancelCondition*.
post	Indicates the *postCondition*.

10.1.85 Context

Represents AML Context (p. 171) metaclass.

Icon: 
Keyword: <<context>>
Applies to: Package

Tags:

situationState: String[0..1]	Specification of the State determining the situation for the Context. Represents the AML meta-association Context::situationState.
situationConstraint: String[0..1]	Specification of the constraint determining the situation for the Context. Represents the AML meta-association Context::situationConstraint.

Constraints:

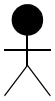
1. The situationState and the situationConstraint tagged values cannot be both specified at once:

   self.situationState->isEmpty() or self.situationContext->isEmpty()

10.1.86 AutonomousActor

Represents the UML Actor extended by the features and applicability of the AutonomousEntityType (p. 24).

This stereotype is optional in the UML 2.0 profile for AML and should be implemented only in the case that the implementation environment (e.g. a modeling CASE tool) does not allow the specification of tagged values for standard UML elements (Actor in this case), but only for stereotypes.

Icon: 
Keyword: <<autonomous>>
Parents: AutonomousEntityType (p. 182)
Applies to: Actor

10.2 UML 1.5 Profile for AML

Each of the following subsections specifies one stereotype defined in the UML 1.5 profile.

10.2.1 EntityType (abstract)

Represents AML EntityType (p. 23) metaclass.

Applies to: Classifier

10.2.2 BehavioralEntityType (abstract)

Represents AML BehavioralEntityType (p. 24) metaclass.

Parents: EntityType (p. 207), BehavioredSemiEntityType (p. 214), and SocializedSemiEntityType (p. 208)
Applies to: Class

10.2.3 AutonomousEntityType (abstract)

Represents AML AutonomousEntityType (p. 24) metaclass.

Parents: BehavioralEntityType (p. 207) and MentalSemiEntityType (p. 227)
Applies to: Class

10.2.4 AgentType

Represents AML AgentType (p. 25) metaclass.

Icon: 
Keyword: <<agent>>
Parents: AutonomousEntityType (p. 207)
Applies to: Class

10.2.5 ResourceType

Represents AML ResourceType (p. 27) metaclass.

Icon: 
Keyword: <<resource>>
Parents: BehavioralEntityType (p. 207)
Applies to: Class

10.2.6 EnvironmentType

Represents AML EnvironmentType (p. 28) metaclass.

Icon: 
Keyword: <<environment>>
Parents: AutonomousEntityType (p. 207)
Applies to: Class

10.2.7 OrganizationUnitType

Represents AML OrganizationUnitType (p. 31) metaclass.

Icon: 
Keyword: <<organization unit>>
Parents: EnvironmentType (p. 208)
Applies to: Class

10.2.8 SocializedSemiEntityType (abstract)

Represents AML SocializedSemiEntityType (p. 33) metaclass.

Applies to: Class
Tags:

supportedAcl: String[0..1]	Comma-separated list of identifiers of supported agent communication languages. Represents the AML meta-attribute SocializedSemiEntityType::supportedAcl.
supportedCl: String[0..1]	Comma-separated list of identifiers of supported content languages. Represents the AML meta-attribute SocializedSemiEntityType::supportedCl.

supportedEncoding: String[0..1]	Comma-separated list of identifiers of supported message content encodings. Represents the AML meta-attribute SocializedSemiEntityType::supportedEncoding.
supportedOntology: String[0..1]	Comma-separated list of identifiers of supported ontologies. Represents the AML meta-attribute SocializedSemiEntityType::supportedOntology.

10.2.9 SocialProperty (abstract)

Represents AML SocialProperty (p. 35) metaclass. Its concrete subclasses represent SocialProperties of different social role kinds.

Parents: ServicedProperty (p. 220)
Applies to: Attribute and AssociationEnd
Constraints:

1. SocialProperty can be owned only by a SocializedSemiEntityType (p. 208) or a SocialAssociation (p. 210):

   self.extendedElement->forAll(ee|
   ((ee.oclIsKindOf(Attribute) and ee.owner->notEmpty()) implies
   ee.owner.oclIsKindOf(SocializedSemiEntityType)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.association.oclIsKindOf(SocialAssociation)))

2. The type or the participant meta-association refers to a SocializedSemiEntityType:

   self.extendedElement->forAll(ee|
   (ee.oclIsKindOf(Attribute) implies
   ee.type.oclIsKindOf(SocializedSemiEntityType)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.participant.oclIsKindOf(SocializedSemiEntityType)))

10.2.10 SocialPropertyPeer

Represents AML SocialProperty (p. 35) metaclass with the socialRole meta-attribute set to *peer*.

Icon: ◭
Keyword: <<peer>>
Parents: SocialProperty (p. 209)
Applies to: Attribute and AssociationEnd

10.2.11 SocialPropertySuperordinate

Represents AML SocialProperty (p. 35) metaclass with the socialRole meta-attribute set to *superordinate*.

Icon: ▲
Keyword: <<super>>
Parents: SocialProperty (p. 209)
Applies to: Attribute and AssociationEnd

10.2.12 SocialPropertySubordinate

Represents AML SocialProperty (p. 35) metaclass with the socialRole meta-attribute set to *subordinate*.

Icon: △
Keyword: <<sub>>
Parents: SocialProperty (p. 209)
Applies to: Attribute and AssociationEnd

10.2.13 SocialAssociation

Represents AML SocialAssociation (p. 38) metaclass.

Keyword: <<social>>
Applies to: Association
Constraints:

1. The connection meta-association refers only to SocialProperties (p. 209):

   self.connection->forAll(oclIsKindOf(SocialProperty))

10.2.14 EntityRoleType

Represents AML EntityRoleType (p. 39) metaclass.

Icon:
Keyword: <<entity role>>
Parents: BehavioredSemiEntityType (p. 214), MentalSemiEntityType (p. 227), and SocializedSemiEntityType (p. 208)
Applies to: Class

10.2.15 RoleProperty

Represents AML RoleProperty (p. 41) metaclass.

Keyword: <<role>>
Applies to: Attribute and AssociationEnd
Constraints:

1. RoleProperty can be owned only by a BehavioralEntityType (p. 207) or a PlayAssociation (p. 211):

   self.extendedElement->forAll(ee|
   ((ee.oclIsKindOf(Attribute) and ee.owner->notEmpty()) implies
   ee.owner.oclIsKindOf(BehavioralEntityType)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.association.oclIsKindOf(PlayAssociation)))

2. The type or the participant meta-association refers to an EntityRoleType (p. 210):

   self.extendedElement->forAll(ee|
   (ee.oclIsKindOf(Attribute) implies
   ee.type.oclIsKindOf(EntityRoleType)) and

(ee.oclIsKindOf(AssociationEnd) implies
ee.participant.oclIsKindOf(EntityRoleType)))

10.2.16 PlayAssociation

Represents AML PlayAssociation (p. 43) metaclass.

Keyword: <<play>>
Applies to: Association
Constraints:

1. PlayAssociation is binary:

   self.connection->size()=2

2. One connection must refer to a RoleProperty (p. 210):

   self.connection->exists(co|co.oclIsKindOf(RoleProperty))

3. The multiplicity of the other connection must be at most 1:

   self.connection->exists(co| not co.oclIsKindOf(RoleProperty) and co.upperbound()<=1)

10.2.17 CreateRoleAction

Represents AML CreateRoleAction (p. 45) metaclass. One of its outgoing object flows represents the created entity role.

Icon: 
Keyword: <<create role>>
Applies to: ActionState
Tags:

player: String[1]	Specification of the player of the created entity role. Represents the AML meta-association CreateRoleAction::player.
roleProperty: String[1]	Specification of the RoleProperty (p. 210) where the created entity role is being placed. Represents the AML meta-association CreateRoleAction::roleProperty.
roleType: String[1]	Specification of the instantiated EntityRoleType (p. 210). Represents the AML meta-association CreateRoleAction::roleType.

Constraints:

1. The player tagged value must refer to an existing behavioral entity (e.g. by its name or an expression evaluated to a behavioral entity). This constraint cannot be expressed in OCL.

2. The roleProperty tagged value must be the name of a RoleProperty owned by the player's type/classifier. This constraint cannot be expressed in OCL.

3. The roleType tagged value must be the name of an existing EntityRoleType. This constraint cannot be expressed in OCL.

4. The type of the ObjectFlowState representing the created entity role must conform to the type specified by the roleType tagged value. This constraint cannot be expressed in OCL.
5. The EntityRoleType referred to by the roleType tagged value must conform to the type of the RoleProperty referred to by the roleProperty tagged value. This constraint cannot be expressed in OCL.

10.2.18 DisposeRoleAction

Represents AML DisposeRoleAction (p. 47) metaclass. It destroys the entity roles represented by its incoming object flows.

Icon: 
Keyword: <<dispose role>>
Applies to: ActionState
Constraints:

1. The DisposeRoleAction specifies a UML DestroyObjectAction:

   self.entry.action.oclIsKindOf(DestroyObjectAction)

10.2.19 AgentExecutionEnvironment

Represents AML AgentExecutionEnvironment (p. 49) metaclass.

Icon: 
Keyword: <<agent execution environment>>
Parents: BehavioredSemiEntityType (p. 214)
Applies to: Node

10.2.20 HostingProperty

Represents AML HostingProperty (p. 51) metaclass.

Icon:  for hosting of an agent,
 for hosting of a resource,
 for hosting of an environment,
 for hosting of an organization unit
Keyword: <<hosting>>
Parents: ServicedProperty (p. 220)
Applies to: Attribute and AssociationEnd
Tags:

hostedAs: String[0..1]	The set of hosting kinds that the owning AgentExecutionEnvironment (p. 212) provides to the HostingProperty's type. The set is specified as a comma-separated list of keywords defined by the HostingKind (p. 54) enumeration from the AML metamodel. Represents the AML meta-attribute HostingProperty::hostingKind.

Constraints:

1. HostingProperty can be owned only by an AgentExecutionEnvironment or a HostingAssociation (p. 213):

   ```
   self.extendedElement->forAll(ee|
   ((ee.oclIsKindOf(Attribute) and ee.owner->notEmpty()) implies
   ee.owner.oclIsKindOf(AgentExecutionEnvironment)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.association.oclIsKindOf(HostingAssociation)))
   ```

2. The type or the participant meta-association refers to an EntityType (p. 207):

   ```
   self.extendedElement->forAll(ee|
   (ee.oclIsKindOf(Attribute) implies
   ee.type.oclIsKindOf(EntityType)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.participant.oclIsKindOf(EntityType)))
   ```

10.2.21 HostingAssociation

Represents AML HostingAssociation (p. 54) metaclass.

Keyword: <<hosting>>
Applies to: Association
Constraints:

1. HostingAssociation is binary:

   ```
   self.connection->size()=2
   ```

2. One connection must refer to a HostingProperty (p. 212):

   ```
   self.connection->exists(co|co.oclIsKindOf(HostingProperty))
   ```

10.2.22 Ontology

Represents AML Ontology (p. 56) metaclass.

Icon: 
Keyword: <<ontology>>
Applies to: Package

10.2.23 OntologyClass

Represents AML OntologyClass (p. 57) metaclass.

Icon: 
Keyword: <<oclass>>
Applies to: Class

10.2.24 OntologyUtility

Represents AML OntologyUtility (p. 58) metaclass.

Icon: 
Keyword: <<outility>>
Applies to: Class

10.2.25 BehavioredSemiEntityType (abstract)

Represents AML BehavioredSemiEntityType (p. 60) metaclass.

Parents: ServicedElement (p. 220)
Applies to: Class

10.2.26 BehaviorFragment

Represents AML BehaviorFragment (p. 65) metaclass.

Icon: 
Keyword: <<behavior fragment>>
Parents: BehavioredSemiEntityType (p. 214)
Applies to: Class

10.2.27 MultiMessage

Represents AML MultiMessage (p. 72) metaclass.

Icon: 
Keyword: <<multi>>
Applies to: Message
Tags:

sendDiscriminator: String[0..1]	The constraint which specifies the MultiMessage senders when it is sent from a ClassifierRole having the multiplicity upper bound greater than 1. Senders are those instances represented by the ClassifierRole for which the sendDiscriminator is evaluated to *true*. Represents the AML meta-association MultiMessage::sendDiscriminator.
receiveDiscriminator: String[0..1]	The constraint which specifies the MultiMessage receivers when it is sent to a ClassifierRole having the multiplicity upper bound greater than 1. Receivers are those instances represented by the ClassifierRole for which the receiveDiscriminator is evaluated to *true*. Represents the AML meta-association MultiMessage::receiveDiscriminator.
toItself: Boolean[1]	If *true*, the MultiMessage is sent also to its sender when the sender belongs to the group of receivers. If *false*, the sender is excluded from the group of receivers. Represents the AML meta-attribute MultiMessage::toItself.

Constraints:

1. The sendDiscriminator tagged value can be specified only if the sender is represented by a ClassifierRole having the multiplicity upper bound greater than 1:

   ```
   self.sendDiscriminator->notEmpty() implies
   self.sender.multiplicity.upperbound>1
   ```

2. The receiveDiscriminator tagged value can be specified only if the receiver is represented by a ClassifierRole having the multiplicity upper bound greater than 1:

   self.receiveDiscriminator->notEmpty() implies
   self.receiver.multiplicity.upperbound>1

10.2.28 DecoupledMessage

Represents AML DecoupledMessage (p. 74) metaclass.

Icon: 
Keyword: <<decoupled>>
Parents: MultiMessage (p. 215)
Applies to: Message
Tags:

payload: String[0..1]	The type of the object transmitted, specified as the name of a DecoupledMessagePayload (p. 215). Represents the AML meta-association DecoupledMessage::payload.

Constraints:

1. The payload tagged value must be the name of an existing DecoupledMessagePayload. This constraint cannot be expressed in OCL.

10.2.29 DecoupledMessagePayload

Represents AML DecoupledMessagePayload (p. 76) metaclass.

Icon: 
Keyword: <<dm payload>>
Applies to: Class

10.2.30 Subset

Represents AML Subset (p. 76) metaclass.

Keyword: <<sub>>
Applies to: Dependency
Constraints:

1. Subset has one client:

   self.client->size()=1

2. The client and the suppliers of a Subset are UML ClassifierRoles:

   self.client->forAll(me|me.oclIsKindOf(ClassifierRole)) and
   self.supplier->forAll(me|me.oclIsKindOf(ClassifierRole))

3. All bases of the supplier ClassifierRoles must conform to the base of the client ClassifierRole:

   Classifier::conformsTo(other: Classifier): Boolean;
   conformsTo = (self=other) or (self.allParents()->includes(other))

self.supplier.base->forAll(oclAsType(Classifier).conformsTo(
self.client.base.oclAsType(Classifier)))

10.2.31 Join

Represents AML Join (p. 78) metaclass.

Keyword: <<join>>
Applies to: Dependency
Tags:

selector: String[0..1]	Specifies the set of instances being joined. Represents the AML meta-association Join::selector.

Constraints:

1. Join has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client and supplier of a Join are UML ClassifierRoles:

   self.client.oclIsKindOf(ClassifierRole) and
   self.supplier.oclIsKindOf(ClassifierRole)

3. The supplier ClassifierRole must have the multiplicity upper bound greater than 1:

   self.supplier.oclAsType(ClassifierRole).multiplicity.upperbound>1

4. The base of the client ClassifierRole must conform to the base of the supplier ClassifierRole:

   -- conformsTo() operation is defined in the Subset stereotype

   self.client.base->forAll(oclAsType(Classifier).conformsTo(
   self.supplier.base.oclAsType(Classifier)))

10.2.32 CreateAttribute

Represents AML AttributeChange's (p. 81) createdLifeline meta-association.

Keyword: <<create attribute>>
Applies to: Dependency
Constraints:

1. CreateAttribute has one client:

   self.client->size()=1

2. The client and the suppliers of a CreateAttribute are UML ClassifierRoles or Instances in a CollaborationInstanceSet:

   (self.client.oclIsKindOf(ClassifierRole) or
   (self.client.oclIsKindOf(Instance) and
   self.client.oclAsType(Instance).collaborationInstanceSet->notEmpty()))
   and (self.supplier.oclIsKindOf(ClassifierRole) or
   (self.supplier.oclIsKindOf(Instance) and
   self.supplier.oclAsType(Instance).collaborationInstanceSet->
   notEmpty()))

3. Each CreateAttribute's supplier represents (a value of) an attribute of a Classifier represented by the CreateAttribute's client. This constraint cannot be expressed in OCL.

10.2.33 DestroyAttribute

Represents AML AttributeChange's (p. 81) destroyedLifeline meta-association.

Keyword: <<destroy attribute>>
Applies to: Dependency
Constraints:

1. DestroyAttribute has one client:

   self.client->size()=1

2. The client and the suppliers of a DestroyAttribute are UML ClassifierRoles or Instances in a CollaborationInstanceSet:

   (self.client.oclIsKindOf(ClassifierRole) or
   (self.client.oclIsKindOf(Instance) and
   self.client.oclAsType(Instance).collaborationInstanceSet->notEmpty()))
   and (self.supplier.oclIsKindOf(ClassifierRole) or
   (self.supplier.oclIsKindOf(Instance) and
   self.supplier.oclAsType(Instance).collaborationInstanceSet->
   notEmpty()))

3. Each CreateAttribute's supplier represents (a value of) an attribute of a Classifier represented by the CreateAttribute's client. This constraint cannot be expressed in OCL.

10.2.34 CommunicationSpecifier (abstract)

Represents AML CommunicationSpecifier (p. 85) metaclass.

Applies to: ModelElement
Tags:

acl: String[0..1]	Denotes the agent communication language in which CommunicationMessages (p. 218) are expressed. Represents the AML meta-attribute CommunicationSpecifier::acl.
cl: String[0..1]	Denotes the language in which the CommunicationMessage's content is expressed, also called the content language. Represents the AML meta-attribute CommunicationSpecifier::cl.
encoding: String[0..1]	Denotes the specific encoding of the CommunicationMessage's content. Represents the AML meta-attribute CommunicationSpecifier::encoding.
ontology: String[0..1]	Comma-separated list of identifiers of ontologies used to give a meaning to the symbols in the CommunicationMessage's content expression. Represents the AML meta-attribute CommunicationSpecifier::ontology.

10.2.35 CommunicationMessage

Represents AML CommunicationMessage (p. 86) metaclass.

Icon: 
Keyword: <<communication>>
Parents: DecoupledMessage (p. 215) and CommunicationSpecifier (p. 217)
Applies to: Message
Tags:

payload: String[0..1]	The type of the object transmitted, specified as the name of a CommunicationMessagePayload (p. 218). Represents the AML meta-association CommunicationMessage::payload.

Constraints:

1. The payload tagged value must be the name of an existing CommunicationMessagePayload. This constraint cannot be expressed in OCL.

10.2.36 CommunicationMessagePayload

Represents AML CommunicationMessagePayload (p. 87) metaclass.

Icon: 
Keyword: <<cm payload>>
Parents: DecoupledMessagePayload (p. 215)
Applies to: Class
Tags:

performative: String[0..1]	Performative of the CommunicationMessagePayload. Represents the AML meta-attribute CommunicationMessagePayload::performative.

10.2.37 CommunicativeInteraction

Represents AML CommunicativeInteraction (p. 89) metaclass. If the CommunicativeInteraction is parametrized, it represents the InteractionProtocol (p. 89) from the AML metamodel.

Keyword: <<communicative>> (but is usually omitted from diagrams)
Parents: CommunicationSpecifier (p. 217)
Applies to: Collaboration and CollaborationInstanceSet

10.2.38 SendDecoupledMessageAction

Represents AML SendDecoupledMessageAction (p. 94) metaclass.

Icon: 
Keyword: <<send decoupled message>>
Applies to: ActionState
Constraints:

1. The SendDecoupledMessageAction specifies a UML AsynchronousInvocationAction:

self.entry.action.oclIsKindOf(AsynchronousInvocationAction)

10.2.39 SendCommunicationMessageAction

Represents AML SendCommunicationMessageAction (p. 95) metaclass.

Icon: 
Keyword: <<send communication message>>
Parents: SendDecoupledMessageAction (p. 218) and CommunicationSpecifier (p. 217)
Applies to: ActionState

10.2.40 AcceptDecoupledMessageAction

Represents AML AcceptDecoupledMessageAction (p. 97) metaclass. Its name represents a DecoupledMessageTrigger (p. 99), and one of its outgoing object flows may represent the object that has been received as a DecoupledMessage (p. 74).

Icon: 
Keyword: <<accept decoupled message>>
Applies to: ActionState
Constraints:

1. The type of the ObjectFlowState representing the received object must be a DecoupledMessagePayload (p. 215). This constraint cannot be expressed in OCL.

10.2.41 AcceptCommunicationMessageAction

Represents AML AcceptCommunicationMessageAction (p. 98) metaclass. Its name represents a CommunicationMessageTrigger (p. 100), and one of its outgoing object flows may represent the object that has been received as a CommunicationMessage (p. 86).

Icon: 
Keyword: <<accept communication message>>
Applies to: ActionState
Constraints:

1. The type of the ObjectFlowState representing the received object must be a CommunicationMessagePayload (p. 218). This constraint cannot be expressed in OCL.

10.2.42 ServiceSpecification

Represents AML ServiceSpecification (p. 103) metaclass.

Icon: 
Keyword: <<service specification>>
Parents: CommunicationSpecifier (p. 217)
Applies to: Class

10.2.43 ProviderParameter

Represents a special TemplateParameter (from UML) used to represent the type of the AML ServiceProtocol::providerParameter meta-association. ServiceProtocol (p. 105) from the AML metamodel is represented in the UML profile as a parametrized CommunicativeInteraction (p. 218) stereotype with its TemplateParameters stereotyped either by the ProviderParameter or the ClientParameter (p. 220) stereotypes.

Keyword: <<provider>>
Applies to: TemplateParameter
Constraints:

1. ProviderParameter can be used only for a CommunicativeInteraction:

   self.template.oclIsKindOf(CommunicativeInteraction)

10.2.44 ClientParameter

Represents a special TemplateParameter (from UML) used to represent the type of the AML ServiceProtocol::clientParameter meta-association. ServiceProtocol (p. 105) from the AML metamodel is represented in the UML profile as a parametrized CommunicativeInteraction (p. 218) stereotype with its TemplateParameters stereotyped either by the ProviderParameter (p. 220) or the ClientParameter stereotypes.

Keyword: <<client>>
Applies to: TemplateParameter
Constraints:

1. ClientParameter can be used only for a CommunicativeInteraction:

   self.template.oclIsKindOf(CommunicativeInteraction)

10.2.45 ServicedElement (abstract)

Represents AML ServicedElement (p. 108) metaclass.

Applies to: ModelElement

10.2.46 ServicedProperty

Represents AML ServicedProperty (p. 109) metaclass.

Keyword: <<serviced>>
Parents: ServicedElement (p. 220)
Applies to: Attribute
Constraints:

1. The type meta-association refers to a BehavioredSemiEntityType (p. 214):

   self.type.oclIsKindOf(BehavioredSemiEntityType)

10.2.47 ServiceProvision

Represents AML ServiceProvision (p. 112) metaclass. Provider template parameter substitutions are specified as the body of the attached UML Comment.

Keyword: <<provides>>
Applies to: Dependency
Constraints:

1. ServiceProvision has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client of a ServiceProvision is a ServicedElement (p. 220) and the supplier of a ServiceProvision is a ServiceSpecification (p. 219):

   self.client.oclIsKindOf(ServicedElement) and
   self.supplier.oclIsKindOf(ServiceSpecification)

3. The specified provider template parameter substitutions bind all (and only) the provider parameters from all service protocols of the ServiceSpecification. This constraint cannot be expressed in OCL.

10.2.48 ServiceUsage

Represents AML ServiceUsage (p. 114) metaclass. Client template parameter substitutions are specified as the body of the attached UML Comment.

Keyword: <<uses>>
Applies to: Usage
Constraints:

1. ServiceUsage has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client of a ServiceUsage is a ServicedElement (p. 220) and the supplier of a ServiceUsage is a ServiceSpecification (p. 219):

   self.client.oclIsKindOf(ServicedElement) and
   self.supplier.oclIsKindOf(ServiceSpecification)

3. The specified client template parameter substitutions bind all (and only) the client parameters from all service protocols of the ServiceSpecification. This constraint cannot be expressed in OCL.

10.2.49 PerceivingAct

Represents AML PerceivingAct (p. 117) metaclass.

Keyword: <<pa>>
Applies to: Operation
Constraints:

1. PerceivingAct can be owned only by a PerceptorType (p. 222):

   self.owner.oclIsKindOf(PerceptorType)

10.2.50 PerceptorType

Represents AML PerceptorType (p. 118) metaclass.

Icon: 
Keyword: <<perceptor type>>
Parents: BehavioredSemiEntityType (p. 214)
Applies to: Class

10.2.51 Perceptor

Represents AML Perceptor (p. 119) metaclass.

Icon: 
Keyword: <<perceptor>>
Parents: ServicedProperty (p. 220)
Applies to: Attribute
Constraints:

1. Perceptor can be owned only by a BehavioredSemiEntityType (p. 214):

   self.owner.oclIsKindOf(BehavioredSemiEntityType)

2. The type meta-association refers to a PerceptorType (p. 222):

   self.type.oclIsKindOf(BehavioredSemiEntityType)

10.2.52 PerceptAction

Represents AML PerceptAction (p. 121) metaclass.

Icon: 
Keyword: <<percept>>
Applies to: ActionState
Constraints:

1. The PerceptAction specifies a UML CallOperationAction:

   self.entry.action.oclIsKindOf(CallOperationAction)

2. The operation meta-association must refer to a PerceivingAct (p. 221):

   self.entry.action.operation.oclIsKindOf(PerceivingAct)

10.2.53 Perceives

Represents AML Perceives (p. 123) metaclass.

Keyword: <<perceives>>
Applies to: Dependency

10.2.54 EffectingAct

Represents AML EffectingAct (p. 123) metaclass.

Keyword: <<ea>>
Applies to: Operation
Constraints:

1. EffectingAct can be owned only by an EffectorType (p. 223):

   self.owner.oclIsKindOf(EffectorType)

10.2.55 EffectorType

Represents AML EffectorType (p. 124) metaclass.

Icon: 
Keyword: <<effector type>>
Parents: BehavioredSemiEntityType (p. 214)
Applies to: Class

10.2.56 Effector

Represents AML Effector (p. 125) metaclass.

Icon: 
Keyword: <<effector>>
Parents: ServicedProperty (p. 220)
Applies to: Attribute
Constraints:

1. Effector can be owned only by a BehavioredSemiEntityType (p. 214):

   self.owner.oclIsKindOf(BehavioredSemiEntityType)

2. The type meta-association refers to an EffectorType (p. 223):

   self.type.oclIsKindOf(EffectorType)

10.2.57 EffectAction

Represents AML EffectAction (p. 126) metaclass.

Icon: 
Keyword: <<effect>>
Applies to: ActionState
Constraints:

1. The EffectAction specifies a UML CallOperationAction:

   self.entry.action.oclIsKindOf(CallOperationAction)

2. The operation meta-association must refer to an EffectingAct (p. 222):

   self.entry.action.operation.oclIsKindOf(EffectingAct)

10.2.58 Effects

Represents AML Effects (p. 127) metaclass.

Keyword: <<effects>>
Applies to: Dependency

10.2.59 Move

Represents AML Move (p. 129) metaclass.

Keyword: <<move>>
Applies to: Dependency
Constraints:

1. Move has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client and supplier of a Move are HostingProperties (p. 212):

   self.client.oclIsKindOf(HostingProperty) and
   self.supplier.oclIsKindOf(HostingProperty)

3. The type or the participant of the HostingProperty at the supplier end must conform to the type or to the participant respectively of the HostingProperty at the client end:

   -- conformsTo() operation is defined in the Subset stereotype

   self.supplier.oclAsType(Attribute).type.
   conformsTo(self.client.oclAsType(Attribute).type or
   self.supplier.oclAsType(AssociationEnd).participant.
   conformsTo(self.client.oclAsType(AssociationEnd).participant

10.2.60 Clone

Represents AML Clone (p. 130) metaclass.

Keyword: <<clone>>
Applies to: Dependency
Constraints:

1. Clone has one client:

   self.client->size()=1

2. The client and the suppliers of a Clone are Attributes having the HostingProperty (p. 212) stereotype:

   (not (oclIsUndefined(self.client.oclAsType(Attribute)) or
   oclIsUndefined(self.supplier->forAll(oclAsType(Attribute))))) and
   self.client.oclIsKindOf(HostingProperty) and
   self.supplier->forAll(oclIsKindOf(HostingProperty))

3. The types of the HostingProperties at the supplier ends must conform to the type of the HostingProperty at the client end:

   -- conformsTo() operation is defined in the Subset stereotype

   self.supplier.oclAsType(Attribute).type->
   forAll(conformsTo(self.client.oclAsType(Attribute).type)

10.2.61 MobilityAction (abstract)

Represents AML MobilityAction (p. 131) metaclass.

Applies to: ActionState
Constraints:

1. The MobilityAction specifies a UML AddAttributeValueAction:

self.entry.action.oclIsKindOf(AddAttributeValueAction)

2. If the type of the InputPin referred to by the value meta-association of the specified AddAttributeValueAction is defined, it must be an EntityType (p. 182):

   self.entry.action.value.type->notEmpty() implies
   self.entry.action.value.type.oclIsKindOf(EntityType)

3. If the type of the InputPin referred to by the object meta-association of the specified AddAttributeValueAction is defined, it must be an AgentExecutionEnvironment (p. 186):

   self.entry.action.object.type->notEmpty() implies
   self.entry.action.object.type.oclIsKindOf(AgentExecutionEnvironment)

4. The attribute meta-association of the specified AddAttributeValueAction must refer to a HostingProperty (p. 212):

   self.entry.action.attribute.oclIsKindOf(HostingProperty)

5. If the type of the InputPin referred to by the object meta-association of the specified AddAttributeValueAction is defined, the HostingProperty referred to by the attribute meta-association must be an owned attribute of that type:

   self.entry.action.object.type->notEmpty() implies
   self.entry.action.object.type.allAttributes->
   includes(self.entry.action.attribute)

10.2.62 MoveAction

Represents AML MoveAction (p. 132) metaclass.

Icon: 
Keyword: <<move>>
Parents: MobilityAction (p. 224)
Applies to: ActionState

10.2.63 CloneAction

Represents AML CloneAction (p. 133) metaclass. One of its OutputPins represents the entity clone.

Icon: 
Keyword: <<clone>>
Parents: MobilityAction (p. 224)
Applies to: ActionState

10.2.64 MentalState (abstract)

Represents AML MentalState (p. 139) metaclass.

Applies to: ModelElement
Tags:

degree: String[0..1]	The degree of a MentalState. Represents the AML meta-attribute MentalState::degree.

10.2.65 MentalClass (abstract)

Represents AML MentalClass (p. 140) metaclass.

Parents: MentalState (p. 225)
Applies to: Class

10.2.66 ConstrainedMentalElement (abstract)

This stereotype is used to define the possibility of specifying MentalConstraints (p. 141) from the AML metamodel as tagged values, but does not directly represent any AML metaclass. It is used as a common superclass to the ConstrainedMentalClass (p. 226) and MentalProperty (p. 227) stereotypes.

Applies to: ModelElement
Tags:

commitCondition: String[0..1]	An assertion identifying the situation under which an autonomous entity commits to the ConstrainedMentalClass (if the precondition also holds). Represents the AML meta-attribute MentalConstraintKind::commitCondition.
preCondition: String[0..1]	The condition that must hold before the ConstrainedMentalClass can become effective (i.e. a goal can be committed to or a plan can be executed). Represents the AML meta-attribute MentalConstraintKind::preCondition.
invariant: String[0..1]	The condition that holds during the period the ConstrainedMentalClass remains effective. Represents the AML meta-attribute MentalConstraintKind::invariant.
cancelCondition: String[0..1]	An assertion identifying the situation under which an autonomous entity cancels attempting to accomplish the ConstrainedMentalClass. Represents the AML meta-attribute MentalConstraintKind::cancelCondition.
postCondition: String[0..1]	The condition that holds after the ConstrainedMentalClass has been accomplished (i.e. a goal has been achieved or a plan has been executed). Represents the AML meta-attribute MentalConstraintKind::postCondition.

10.2.67 ConstrainedMentalClass (abstract)

Represents AML ConstrainedMentalClass (p. 140) metaclass.

Parents: MentalClass (p. 226) and ConstrainedMentalElement (p. 226)
Applies to: Class

10.2.68 MentalRelationship (abstract)

Represents AML MentalRelationship (p. 143) metaclass.

Parents: MentalState (p. 225)
Applies to: ModelElement

10.2.69 MentalSemiEntityType (abstract)

Represents AML MentalSemiEntityType (p. 143) metaclass.

Applies to: Class

10.2.70 MentalProperty

Represents AML MentalProperty (p. 144) metaclass.

Keyword: <<mental>>
Parents: ConstrainedMentalElement (p. 226)
Applies to: Attribute and AssociationEnd
Tags:

degree: String[0..1]	The degree of the MentalClass (p. 226) specified as the type of the MentalProperty. If specified, it overrides the degree tagged value specified for the MentalClass itself. Represents the AML meta-attribute MentalProperty::degree.

Constraints:

1. MentalProperty can be owned only by a MentalSemiEntityType (p. 227) or a MentalAssociation (p. 228):

   ```
   self.extendedElement->forAll(ee|
   ((ee.oclIsKindOf(Attribute) and ee.owner->notEmpty()) implies
   ee.owner.oclIsKindOf(MentalSemiEntityType)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.association.oclIsKindOf(MentalAssociation)))
   ```

2. The type or the participant meta-association refers to a MentalClass:

   ```
   self.extendedElement->forAll(ee|
   (ee.oclIsKindOf(Attribute) implies
   ee.type.oclIsKindOf(MentalClass)) and
   (ee.oclIsKindOf(AssociationEnd) implies
   ee.participant.oclIsKindOf(MentalClass)))
   ```

3. The MentalClass referred to by the type or the participant meta-association cannot be a Plan (p. 229):

   ```
   self.extendedElement->forAll(ee|
   (ee.oclIsKindOf(Attribute) implies
   (not ee.type.oclIsKindOf(Plan))) and
   ```

(ee.oclIsKindOf(AssociationEnd) implies
(not ee.participant.oclIsKindOf(Plan))))

10.2.71 MentalAssociation

Represents AML MentalAssociation (p. 147) metaclass.

Keyword: <<mental>>
Applies to: Association
Constraints:

1. MentalAssociation is binary:

   self.connection->size()=2

2. One connection must refer to a MentalProperty (p. 227):

   self.connection->exists(co|co.oclIsKindOf(MentalProperty))

10.2.72 Responsibility

Represents AML Responsibility (p. 148) metaclass.

Keyword: <<responsible>>
Applies to: Dependency
Constraints:

1. The suppliers of a Responsibility are MentalClasses (p. 226):

   self.supplier->forAll(oclIsKindOf(MentalClass))

10.2.73 Belief

Represents AML Belief (p. 149) metaclass.

Icon: 
Keyword: <<belief>>
Parents: MentalClass (p. 226)
Applies to: Class
Tags:

constraint: String[0..1]	The specification of the information a Belief represents. Represents the AML meta-attribute Belief::constraint.

10.2.74 Goal (abstract)

Represents AML Goal (p. 151) metaclass.

Parents: ConstrainedMentalClass (p. 226)
Applies to: Class

10.2.75 DecidableGoal

Represents AML DecidableGoal (p. 152) metaclass.

Icon: 
Keyword: <<dgoal>>
Parents: Goal (p. 228)
Applies to: Class

10.2.76 UndecidableGoal

Represents AML UndecidableGoal (p. 154) metaclass.

Icon: 
Keyword: <<ugoal>>
Parents: Goal (p. 228)
Applies to: Class

10.2.77 Plan

Represents AML Plan (p. 156) metaclass.

Icon: 
Keyword: <<plan>>
Parents: ConstrainedMentalClass (p. 226)
Applies to: ActivityGraph
Constraints:

1. If the context (see UML StateMachine::context meta-association) for a Plan is specified, it must be a MentalSemiEntityType (p. 227):

   ```
   self.context->notEmpty() implies
   self.context.oclIsKindOf(MentalSemiEntityType)
   ```

10.2.78 CommitGoalAction

Represents AML CommitGoalAction (p. 158) metaclass. One of its outgoing object flows represents the created Goal (p. 151) instance.

Icon: 
Keyword: <<commit goal>>
Applies to: ActionState
Tags:

mentalSemiEntity: String[1]	Specification of the mental semi-entity committed to the Goal (p. 228). Represents the AML meta-association CommitGoalAction::mentalSemiEntity.
mentalProperty: String[1]	The name of the MentalProperty (p. 227) where the created Goal instance is being placed. Represents the AML meta-association CommitGoalAction::mentalProperty.
goalType: String[1]	The name of the instantiated Goal. Represents the AML meta-association CommitGoalAction::goalType.

Constraints:

1. The mentalSemiEntity tagged value must refer to an existing mental semi-entity (e.g. by its name or an expression evaluated to a mental semi-entity). This constraint cannot be expressed in OCL.
2. The mentalProperty tagged value must be the name of a MentalProperty owned by the mentalSemiEntity's type/classifier. This constraint cannot be expressed in OCL.
3. The goalType tagged value must be the name of an existing Goal. This constraint cannot be expressed in OCL.
4. The type of the ObjectFlowState representing the created Goal instance must conform to the type specified by the goalType tagged value. This constraint cannot be expressed in OCL.
5. The Goal referred to by the goalType tagged value must conform to the type of the MentalProperty referred to by the mentalProperty tagged value. This constraint cannot be expressed in OCL.

10.2.79 CancelGoalAction

Represents AML CancelGoalAction (p. 160) metaclass. It destroys the Goal (p. 151) instances represented by its incoming object flows.

Icon: 
Keyword: <<cancel goal>>
Applies to: ActionState
Constraints:

1. The CancelGoalAction specifies a UML DestroyObjectAction:

   self.entry.action.oclIsKindOf(DestroyObjectAction)

10.2.80 Contribution

Represents AML Contribution (p. 162) metaclass. It is represented by a binary UML Dependency relationship. A Contribution's contributor is represented by the Dependency client and the beneficiary by the Dependency supplier.

Alternatively, the Contribution relationship from the AML metamodel can be represented by UML Association with a stereotype representing its kind (<<sufficient>>, <<necessary>>, or <<iff>>; see the ContributionKind, p. 169 enumeration for details), and its tagged values contributorConstraintKind and beneficiaryConstraintKind represented as association end properties (with the keywords: *commit*, *pre*, *commpre*, *inv*, *cancel*, or *post*; see the MentalConstraintKind, p. 142 enumeration for details).

Keyword: <<contributes>>
Parents: MentalRelationship (p. 227)
Applies to: Dependency

Tags:

kind: ContributionKind[1]	Determines whether the contribution of the client's tagged value representing a MentalConstraint (p. 141) (from the AML Metamodel) of a specified kind is a necessary, sufficient, or equivalent condition for the supplier's tagged value representing a MentalConstraint (from the AML Metamodel) of a specified kind. Represents the AML meta-association Contribution::kind.
contributorConstraintKind: MentalConstraintKind [0..1]	The kind of the client's tagged value representing a MentalConstraint (from the AML Metamodel) which contributes to the kind of the supplier's tagged value representing a MentalConstraint (from the AML Metamodel). Represents the AML meta-association Contribution::contributorConstraintKind.
beneficiaryConstraintKind: MentalConstraintKind [0..1]	The kind of the supplier's tagged value representing a MentalConstraint (from the AML Metamodel) to which the kind of the client's tagged value representing a MentalConstraint (from the AML Metamodel) contributes. Represents the AML meta-association Contribution::beneficiaryConstraintKind.

Constraints:

1. Contribution has one client and one supplier:

   self.client->size()=1 and self.supplier->size()=1

2. The client and supplier of a Contribution are MentalStates (p. 225):

   self.client.oclIsKindOf(MentalState) and
   self.supplier.oclIsKindOf(MentalState)

3. If the MentalState referred to by the client meta-association is a Belief (p. 228) or a Contribution, the contributorConstraintKind tagged value is unspecified:

   (self.client.oclIsKindOf(Belief) or
   self.client.oclIsKindOf(Contribution)) implies
   self.contributorConstraintKind=#unspecified

4. If the MentalState referred to by the supplier meta-association is a Belief or a Contribution, the beneficiaryConstraintKind tagged value is unspecified:

   (self.supplier.oclIsKindOf(Belief) or
   self.supplier.oclIsKindOf(Contribution)) implies
   self.beneficiaryConstraintKind=#unspecified

10.2.81 ContributionKind (enumeration)

Represents AML ContributionKind (p. 169) metaclass.

Enumeration values:

sufficient (default)	The contributor or its tagged value representing a MentalConstraint (p. 141) (from the AML Metamodel) of the given kind (if specified) is a sufficient condition for the beneficiary or its tagged value representing a MentalConstraint (from the AML Metamodel) of the given kind (if specified). Represents the enumeration value ContributionKind::sufficient from the AML metamodel.
necessary	The contributor or its tagged value representing a MentalConstraint (from the AML Metamodel) of the given kind (if specified) is a necessary condition for the beneficiary or its tagged value representing a MentalConstraint (from the AML Metamodel) of the given kind (if specified). Represents the enumeration value ContributionKind::necessary from the AML metamodel.
iff	The contributor and beneficiary or their tagged value representing MentalConstraints (from the AML Metamodel) of the given kinds (if specified) are equivalent. Represents the enumeration value ContributionKind::iff from the AML metamodel.

10.2.82 MentalConstraintKind (enumeration)

Represents AML MentalConstraintKind (p. 142) metaclass. Its values indicate usage of the MentalConstraints (p. 141) from the AML metamodel specified as tagged values of elements stereotyped by any concrete ConstrainedMentalElement (p. 226) in a relationship stereotyped by Contribution (p. 230).

Enumeration values:

unspecified (default)	Mental constraint kind is unspecified.
commit	Indicates the *commitCondition*.
pre	Indicates the *preCondition*.
commpre	Indicates an AND-ed combination of *commitCondition* and *preCondition*.
inv	Indicates the *invariant*.
cancel	Indicates the *cancelCondition*.
post	Indicates the *postCondition*.

10.2.83 Context

Represents AML Context (p. 171) metaclass.

Icon: 
Keyword: <<context>>
Applies to: Package

Tags:

situationState: String[0..1]	Specification of the State determining the situation for the Context. Represents the AML meta-association Context::situationState.
situationConstraint: String[0..1]	Specification of the constraint determining the situation for the Context. Represents the AML meta-association Context::situationConstraint.

Constraints:

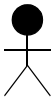
1. The situationState and the situationConstraint tagged values cannot be both specified at once:

   self.situationState->isEmpty() or self.situationContext->isEmpty()

10.2.84 AutonomousActor

Represents the UML Actor extended by the features and applicability of the AutonomousEntityType (p. 24).

This stereotype is optional in the UML 1.5 profile for AML and should be implemented only in the case that the implementation environment (e.g. a modeling CASE tool) does not allow the specification of tagged values for standard UML elements (Actor in this case), but only for stereotypes.

Icon: 
Keyword: <<autonomous>>
Parents: AutonomousEntityType (p. 207)
Applies to: Actor

11 Extension of OCL

New operators

Formal models of MAS usually use different types of modal family logics to describe the system. Our intention is to allow such expressions to be used within OCL expressions.

AML defines a set of operators used to extend the OCL Standard Library [51] to include expressions belonging to modal logic, deontic logic, temporal logic, dynamic logic, epistemic logic, BDI logic, etc. For details see [71] and [75].

Tab. 11-1 summarizes operators added to OCL.

Operator	***Semantics***	***Known as***
Modal Logic		
possible(*p : Boolean*) *: Boolean*	*p* is possible.	$\Diamond p$
necessary(*p : Boolean*) *: Boolean*	*p* is necessary.	$\Box p$
Deontic Logic		
obliged(*p : Boolean*) *: Boolean*	*p* is obliged.	Obl*p*
permitted(*p : Boolean*) *: Boolean*	*p* is permitted.	Per*p*
Temporal Logic		
until(*p : Boolean,q : Boolean*) *: Boolean*	*p* holds until *q* is valid.	*p*U*q*
past(*p : Boolean*) *: Boolean*	*q* held in past.	P*q*
future(*p : Boolean*) *: Boolean*	*p* holds sometimes in the future.	F*p*
afuture(*p : Boolean*) *: Boolean*	*p* holds always in the future.	G*p*
next(*p : Boolean*) *: Boolean*	*p* holds in the next moment.	X*p*, O*p*
Dynamic Logic and KARO		
e.ability(*c : Capability*) *: Boolean*	Mental semi-entity *e* is able to perform the capability *c*.	$A_e c$
e.opportunity(*c : Capability, p : Boolean*) *: Boolean*	Mental semi-entity *e* has the opportunity to perform the capability *c* and so leads to *p*.	$<do_e c>p$
e.possibility(*c : Capability, p : Boolean*) *: Boolean*	If the opportunity to do *c* is indeed present, doing so leads to *p*.	$[do_e c]p$
e.pracPoss(*c : Capability, p : Boolean*) *: Boolean*	Mental semi-entity *e* has the opportuniti and the ability to perform *c*, and doing so leads to *p*.	$PracPoss_e(c, p)$
e.can(*c : Capability, p : Boolean*) *: Boolean*	Mental semi-entity *e* knows that performing *c* constitutes a practical possibility to bring about *p*.	$Can_e(c, p)$
Epistemic Logic		
e.knows(*p : Boolean*) *: Boolean*	Mental semi-entity *e* knows that *p* is true.	*e*K*p*, $K_e\, p$
G.cknows(*p : Boolean*) *: Boolean*	*p* is a common knowledge among mental semi-entities in the set *G*.	$E_G\, p$

Tab. 11-1 New OCL operators

Operator	***Semantics***	***Known as***
e.uncertain(*p : Boolean*) *: Boolean*	Mental semi-entity *e* is uncertain of the truth of *p*. *e* neither believes *p* nor its negation, but believes that *p* is more likely to be true than its negation.	
BDI Logic		
e.believes(*p : Boolean*) *: Boolean*	Mental semi-entity *e* believes that *p* is true.	*e*Bel*p*, $Bel_e\ p$
e.desires(*p : Boolean*) *: Boolean*	Mental semi-entity *e* desires *p* to become true.	*e*Des*p*
e.intends(*p : Boolean*) *: Boolean*	Mental semi-entity *e* intends *p* to become true and will plan to bring it about.	*e*Int*p*
Other operators		
e.happens(*c : Capability*) *: Boolean*	Mental semi-entity *e* currently preforms the capability *c*.	
e.done(*c : Capability*) *: Boolean*	Mental semi-entity *e* has just finished the execution of the capability *c*.	
e.achieved(*g : Goal*) *: Boolean*	Mental semi-entity *e* has just achieved the goal *g*.	

Tab. 11-1 New OCL operators

Examples In the following example the expression utilizing the extension to OCL has been used to specify that possession of the ball can lead to scoring a goal.

```
(TeamHasBall.post) implies (possible(ScoreGoal.post))
```

DecidableGoals (p. 152) TeamHasBall and ScoreGoal are described in Fig. 6-41.

Another example states that if an agent believes that its team does not possess the ball, it switches to the defending strategy.

```
(self.believe(not TeamHasBall.post)) implies
(next(self.happens(Strategy::defend())))
```

BehaviorFragment (p. 65) Strategy and its Capability (p. 62) defend() is described in Fig. 5-7.

Appendix A References

[1] E. Alencar, J. Castro, G. Cysneiros, and J. Mylopoulos. From early requirements modeled by the i* technique to later requirements modeled in precise UML.
In Anais do III Workshop em Engenharia de Requisitos, pages 92–109, Rio de Janeiro, Brazil, July 2000.

[2] A.I. Anton. Goal Identification and Refinement in the Specification of Software-Based Information Systems.
PhD thesis, Georgia Institute of Technology, Atlanta, GA, June 1997.

[3] AUML web page.
http://www.auml.org

[4] B. Bauer. UML class diagrams: Revisited in the context of agent-based systems.
In M.Wooldridge, G.Weiss, and P. Ciancarini, editors, Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), pages 101–118, Montreal, Canada, May 2001. Springer.

[5] B. Bauer, J.P. Müller, and J. Odell. An extension of UML by protocols for multiagent interaction.
In Proceedings of the International Conference on MultiAgent Systems (ICMAS'00), pages 207–214, Boston, MA, USA, July 2000.

[6] B. Bauer, J.P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent interaction.
In P. Ciancarini and M.Wooldridge, editors, Agent-Oriented Software Engineering, pages 91–103. Springer-Verlag, 2001.

[7] C. Bernon, V. Camps, M.P. Gleizes, and G. Picard. Designing agents' behaviours and interactions within the framework of ADELFE methodology.
In Proceedings of the Fourth International Workshop: Engineering Societies in the Agents World (ESAW'03), pages 156–169, Imperial College London, UK, October 2003. Springer-Verlag.

[8] P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. TROPOS: An agent-oriented software development methodology.
Autonomous Agents and Multi-Agent Systems, 2(3):203–236, May 2004.

[9] B. Burmeister. Models and methodology for agent-oriented analysis and design.
In K. Fischer, editor, Working notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems, June 1996.

[10] G. Bush, S. Cranefield, and M. Purvis. The Styx agent methodology.
Technical Report 02, University of Otago, Dunedin, New Zealand, 2001.

[11] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. Non-Functional Requirements in Software Engineering.
Kluwer Academic Publishing, 2000.

[12] M. Cossentino and C. Potts. A CASE tool supported methodology for the design of multi-agent systems.
In Proceedings of the 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, NV, USA, June 2002.

[13] M. Cossentino, L. Sabatucci, and A. Chella. A possible approach to the development of robotic multi-agent systems.
In IEEE/WIC Conference on Intelligent Agent Technology (IAT'03), Halifax, Canada, October 2003.

[14] S. Cranefield, S. Haustein, and M. Purvis. UML-based ontology modelling for software agents.
In Proceedings of the Workshop on Ontologies in Agent, 2001, May 2001.

[15] DAML web page.
http://www.daml.org

[16] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition.
Science of Computer Programming, 20:3–50, 1993.

[17] S.A. DeLoach. Multiagent systems engineering: A methodology and language for designing agent systems.
In Proceedings of the Agent-Oriented Information Systems '99 (AOIS'99), Seattle, WA, May 1999.

[18] S.A. DeLoach, M.F. Wood, and C. H. Sparkman. Multiagent systems engineering.
International Journal of Software Engineering and Knowledge Engineering, 11(3):231–258, 2001.

[19] R. Depke, R. Heckel, and J.M. Kuster. Improving the agent-oriented modeling process by roles.
In Proceedings of the Fifth International Conference on Autonomous Agents, pages 640–647, Montreal, Canada, May/June 2001. ACM.

[20] M. d'Inverno and M. Luck. Understanding Agent Systems.
Springer-Verlag, 2001.

[21] R. Evans, P. Kearny, J. Stark, G. Caire, F. Garijo, G.J. Sanz, F. Leal, P. Chainho, and P. Massonet. MESSAGE: Methodology for engineering systems of software agents. Methodology for agent-oriented software engineering.
Technical Report Eurescom project P907, EDIN 0223-0907, EURESCOM, September 2001.

[22] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein. OIL in a nutshell.
In Dieng, R. et al., editor, Knowledge Acquisition, Modeling, and Management. Proceedings of the European Knowledge Acquisition Conference (EKAW-2000). Lecture Notes in Artificial Intelligence, LNAI, Springer-Verlag, October 2000.

[23] R. Fikes and D.L. McGuinness. An axiomatic semantics for RDF, RDF-S, and DAML+OIL.
URL: http://www.w3.org/TR/2001/NOTE-daml+oil-axioms-20011218, December 2001.

[24] FIPA Specification Repository web page.
http://www.fipa.org/repository/

[25] S. Flake, C. Geiger, and J.M. Küster. Towards UML-based analysis and design of multi-agent systems.
In Proceedings of International NAISO Symposium on Information Science Innovations in Engineering of Natural and Artificial Intelligent Systems (ENAIS'2001), pages 695–701, Dubai, March 2001. ICSC Academic Press.

[26] M.L. Ginsberg. Knowledge interchange format: The KIF of death.
AI Magazine, 12(3):57–63, 1991.

[27] N. Glaser. Conceptual modelling of multi-agent systems (the CoMoMAS engineering environment).
In Kluwer Series on Multiagent Systems, Artificial Societies, and Simulated Organizations, volume 4. Kluwer, May 2002.

[28] Goal Oriented Requirement Language (GRL) web page.
http://www.cs.toronto.edu/km/GRL

[29] J.J. Gomez-Sanz and R. Fuentes. Agent oriented software engineering with INGENIAS.
In Fourth Iberoamerican Workshop on Multi-Agent Systems (Iberagents 2002)—Agent Technology and Software Engineering, Spain, November 2002. University of Malaga.

[30] O. Gutknecht, J. Ferber, and F. Michel. Integrating tools and infrastructures for generic multi-agent systems.
In Proc. of the Fifth International Conference on Autonomous Agents (AA 2001), Montral, Quebec, Canada, May 2001.

[31] M.P. Huget, I. Reinharts-Berger, D. Dori, O. Shehory, and A. Sturm. Modeling-notation source: OPM/MAS. URL:
http://www.auml.org/auml/documents/OPM.pdf.

[32] C. Iglesias, M. Garijo, J.C. Gonzalez, and J.R. Velasco. Analysis and design of multiagent systems using MAS-CommonKADS.
In M.P. Singh, A. Rao, and M.J. Wooldridge, editors, Intelligent Agents IV (LNAI Vol. 1365), volume 1365, pages 313–326. Springer-Verlag, 1998.

[33] N. R. Jennings. On agent-based software engineering.
Artificial Intelligence, 117(2):277–296, 2000.

[34] N.R. Jennings and M. Wooldridge. Software agents.
IEEE Review, 42(1):17–21, January 1996.

[35] C.M. Jonker, M. Klush, and J. Treur. Design of collaborative information agents.
In M. Klush and L. Kerschberg, editors, Cooperative Information Agents IV. Proceedings of CIA 2000., pages 262–283. Springer-Verlag, July 2000.

[36] T. Juan, A. Pearce, and L. Sterling. ROADMAP: Extending the Gaia methodology for complex open systems.
In Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), Bologna, Italy, July 2002.

[37] M. Kang, L. Wang, and K. Taguchi. Modelling mobile agent applications in UML 2.0 activity diagrams.
In Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, SELMAS'2004, pages 104–111, Edinburg, United Kingdom, May 2004.

[38] E.A. Kendall, M.T. Malkoun, and C. Jiang. A methodology for developing agent-based systems for enterprise integration.
In C. Zhang and D. Luckose, editors, Proceedings of the First Australian Workshop on DAI. Lecture Notes on Artificial Intelligence, Canberra, ACT, Australia, November 1995. Springer-Verlag.

[39] D. Kinny and M. Georgeff. Modelling and design of multiagent systems.
In J.P. Müller, M.J. Wooldridge, and N.R. Jennings, editors, Intelligent Agents III: Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL-96), LNAI 1193. Springer-Verlag, August 1996.

[40] J.L. Koning, M.P. Huget, J. Wei, and X. Wang. Extended modeling languages for interaction protocol design.
In M. Wooldridge, G. Weiss, and P. Ciancarini, editors, Proceedings of the Second International Work shop On Agent-Oriented Software Engineering (AOSE-2001), pages 93–100, Montreal, Canada, May 2001. Springer.

[41] J. Lind. MASSIVE: Software Engineering for Multiagent Systems.
PhD thesis, University of the Saarland, 2000.

[42] L. Liu and E. Yu. From requirements to architectural design using goals and scenarios.
In Software Requirements to Architectures Workshop (STRAW 2001), Toronto, Canada, May 2001.

[43] D. Martin (ed.). OWL-S 1.0 release.
URL: http://www.daml.org/services/, November 2003.

[44] MESSAGE web page.
http://www.eurescom.de/public/projects/P900-series/p907

[45] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach.
IEEE Trans. on Sofware Engineering, 18 No. 6:483–497, June 1992.

[46] J. Odell, H.V.D. Parunak, and B. Bauer. Extending UML for agents.
In G. Wagner, Y. Lesperance, and E. Yu, editors, Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence, pages 3–17, Austin, Texas, July 2000. ICue Publishing.

[47] J. Odell, H.V.D. Parunak, and B. Bauer. Representing agent interaction protocols in UML.
In P. Ciancarini and M.Wooldridge, editors, Proceedings on the First International Workshop on Agent-Oriented Software Engineering (AOSE 2000), pages 121–140, Limerick, Ireland, June 2000. Springer.

[48] J. Odell, H.V.D. Parunak, M. Fleischer, and S. Brueckner. Modeling agents and their environment.
In Proceedings of AOSE 2002, pages 16–31, Bologna, Italy, July 2002. Springer.

[49] OIL web page.
http://www.ontoknowledge.org/oil

[50] OMG. Meta Object Facility (MOF) specification.
Version 1.4, formal/2002-04-03, april 2002.

[51] OMG. UML 2.0 OCL specification.
ptc/03-10-14, October 2003.

[52] OMG. Unified Modeling Language specification.
version 1.5. formal/03-03-01, March 2003.

[53] OMG. Unified Modeling Language: Superstructure.
version 2.0. ptc/03-08-02, August 2003.

[54] A. Omicini. Societies and infrastructures in the analysis and design of agent based systems.
In P. Ciancarini and M. Wooldridge, editors, Agent-Oriented Software Engineering Proceedings of the First International Workshop (AOSE-2000), pages 185–194, Limerick, Ireland, June 2000. Springer-Verlag.

[55] OWL web page.
http://www.w3.org/TR/2002/WD-owl-guide-20021104

[56] L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents.
In Proceedings of the First Intemational Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), Bologna, Italy, July 2002.

[57] L. Padgham and M. Winikoff. Developing Intelligent Agent Systems. A practical guide.
John Wiley & Sons Ltd, 2004.

[58] H.V.D. Parunak and J.J. Odell. Represening social structures in UML.
In M. Wooldridge, G. Weiss, and P. Ciancarini, editors, Proceedings of the Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), pages 17–31, Montreal, Canada, May 2001. Springer.

[59] J. Pena, R. Corchuelo, and J. Arjona. A top-down approach for MAS protocol descriptions.
In Proceedings of the 2003 ACM Symposium on Applied Computing (SAC 2003), pages 45–49, Melbourne, FL, USA, March 2003. ACM.

[60] A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture.
In J.F. Allen, R. Fikes, and E. Sandewall, editors, KR'91: Principles of Knowledge Representation and Reasoning, pages 473–484. Morgan Kaufmann, San Mateo, California, 1991.

[61] IBM Rational Rose web page.
http://www.rational.com/rose

[62] L. Shan and H. Zhu. CAMLE: A caste-centric agent modelling language and environment.
In Proceedings of the 3rd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, SELMAS'2004, Edinburg, United Kingdom, May 2004.

[63] V. Silva, A. Garcia, A. Brandao, C. Chavez, C. Lucena, and P. Alencar. Taming agents and objects in software engineering.
In A. Garcia, C. Lucena, J. Castro, A. Omicini, and F. Zambonelli, editors, Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications, volume LNCS 2603, pages 1–25. Springer-Verlag, April 2003.

[64] M.K. Smith, D. McGuinness, R. Volz, and C. Welty. Web Ontology Language (OWL), guide version 1.0, W3C working draft.
URL: http://www.w3.org/TR/2002/WD-owl-guide-20021104, November 2002.

[65] C.H. Sparkman, S.A. DeLoach, and A.L. Self. Automated derivation of complex agent architectures from analysis specifications.
In M.Wooldridge, G.Weiss, and P. Ciancarini, editors, Proceedings of the Second International Workshop On Agent-Oriented Software Engineering (AOSE-2001), pages 77–84, Montreal, Canada, May 2001. Springer.

[66] A. Sturm, D. Dori, and O. Shehory. Single-model method for specifying multi-agent systems.
In 2nd International Joint Conference on Autonomous Agents and Multi-agent Systems (AAMAS-2003), Melbourne, Australia, July 2003.

[67] The Foundation for Intelligent Physical Agents. FIPA specifications repository.
URL: http://www.fipa.org/repository/index.html, 2004.

[68] Tropos web page.
http://www.cs.toronto.edu/km/tropos

[69] W.M. Turski and T.S.E. Maibaum. The Specification of Computer Programs.
Addison-Wesley, 1987.

[70] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective.
In ICSE 2000: 22nd International Conference on Software Engineering, pages 5–19. ACM Press, June 2000.

[71] B. van Linder, J.J. Ch. Meyer, and W. van der Hoek. Formalizing motivational attitudes of agents using the karo framework.
UU-CS (Ext. r. no. 1997-06), Utrecht, the Netherlands: Utrecht University: Information and Computing Sciences, 1997.

[72] W3C. Web Services Activity page.
http://www.w3.org/2002/ws

[73] G. Wagner. A UML Profile for external AOR models.
In Proceedings of the Third International Workshop on Agent-Oriented Software Engineering (AOSE-2002), pages 99–110, Bologna, Italy, July 2002. Springer-Verlag LNAI 2585.

[74] G.Wagner. The Agent-Object-Relationship meta-model: Towards a unified conceptual view of state and behavior.
Information Systems, 28(5):475–504, 2003.

[75] G.Weiss. Multiagent Systems—A Modern Approach to Distributed Artificial Intelligence.
The MIT Press, 3rd edition, 2001.

[76] M. Wooldridge and P. Ciancarini. Agent-oriented software engineering: The state of the art.
In Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Co., 2001.

[77] M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia methodology for agent-oriented analysis and design.
Journal of Autonomous Agents and Multi-Agent Systems, 3(3):285–312, 2000.

[78] Q. Yan, L. Shan, X. Mao, and Z. Qi. RoMAS: A role-based modeling method for multi-agent system.
In J.P. Li, J. Liu, N. Zhong, J. Yen, and J. Zhao, editors, Proceedings of the Second Internacional Conference on Active Media Technology (ICANT2003), pages 156–161, Chongqing, China, May 2003. Chinese Electronical Industry Society, Logistical Engineering University, World Scientific.

[79] E. Yu. Modelling Strategic Relationships for Process Reengineering.
PhD thesis, Department of Computer Science, University of Toronto, Canada, 1995.

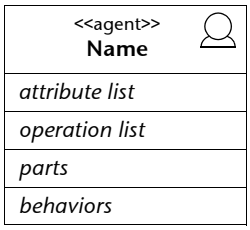
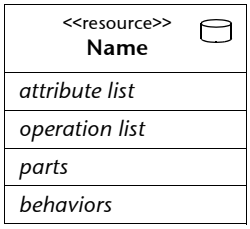
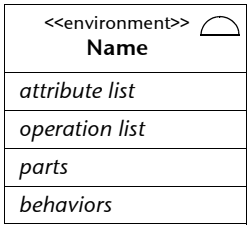
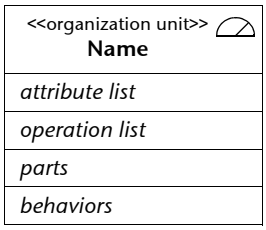
[80] E. Yu. Towards modelling and reasoning support for early-phase requirements engineering.
In Proceedings of IEEE International Symposium on Requirements Engineering RE'97, Washington DC, January 1997. IEEE.

[81] F. Zambonelli, N.R. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia methodology.
ACM Trans on Software Engineering and Methodology, 12(3):317–370, 2003.

[82] H. Zhu. Developing formal specifications of MAS in SLABS: A case study of evolutionary multi-agent ecosystem.
In P. Giorgini, Y. Lesperance, G. Wagner, and E.S.K. Yu, editors, Agent-Oriented Information Systems (AOIS '02). Proceedings of the Fourth International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS-2002 at AAMAS-02), Bologna, Italy, July 2002.

Appendix B AML Notation Summary

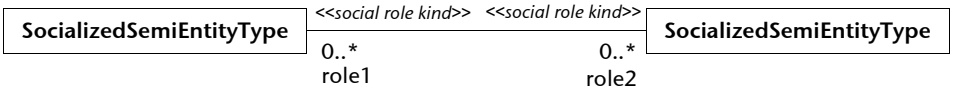
In this section the table summarizing the notation of all AML modeling elements is presented.

AgentType (p. 25)	
ResourceType (p. 27)	
EnvironmentType (p. 28)	
OrganizationUnitType (p. 31)	

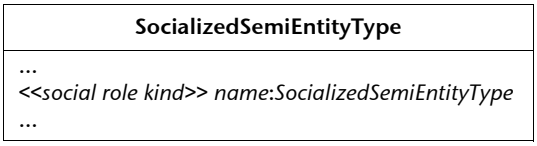
Tab. 11-2 AML notation summary

SocialProperty (p. 35)

When shown as an association end:



When shown as an attribute:



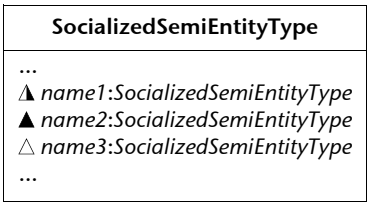
A presentation option when shown as an association end:

(a) peer:

(b) superordinate:

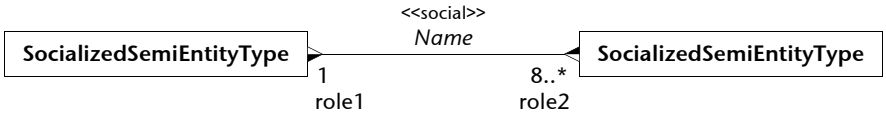
(c) subordinate:

A presentation option when shown as an attribute:

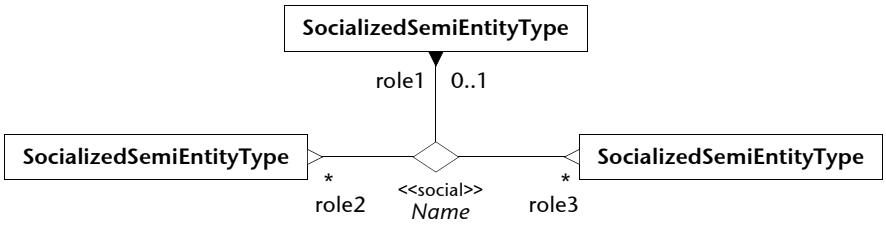


SocialAssociation (p. 38)

Notation of a binary association:

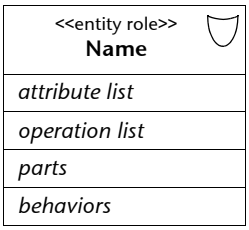


Notation of an n-ary association:



Tab. 11-2 AML notation summary

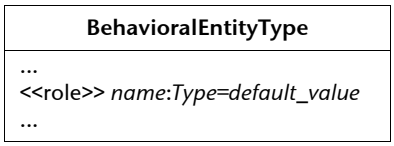
EntityRoleType (p. 39)



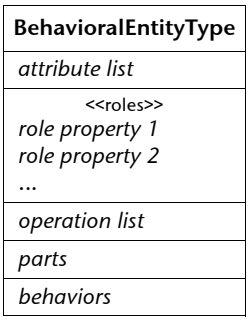
RoleProperty (p. 41)

When shown as the end of a PlayAssociation, the RoleProperty is depicted as a UML association end (see notation of the PlayAssociation, p. 245).

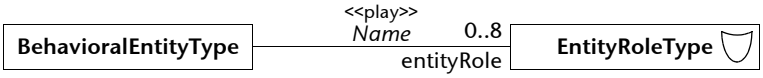
When shown as an attribute:



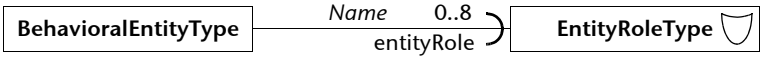
Presentation option:



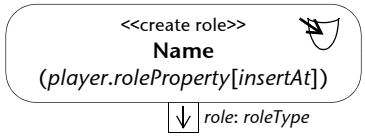
PlayAssociation (p. 43)



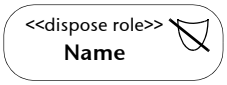
Presentation option:



CreateRoleAction (p. 45)

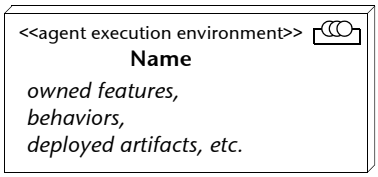


DisposeRoleAction (p. 47)



Tab. 11-2 AML notation summary

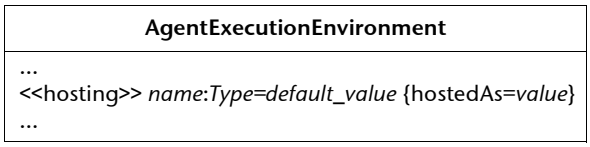
AgentExecutionEnvironment (p. 49)



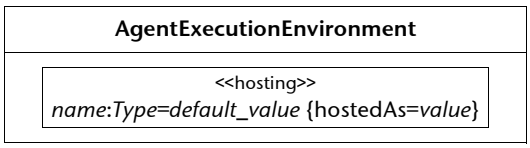
HostingProperty (p. 51)

When shown as the end of a HostingAssociation, the HostingProperty is depicted as a UML association end.

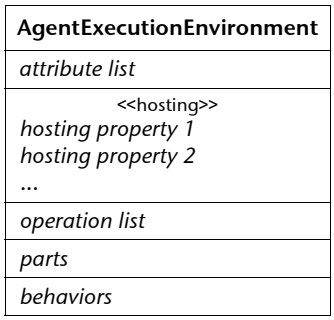
When shown as an attribute:



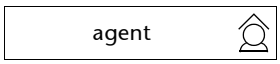
When shown as a connectable element:



Presentation options:



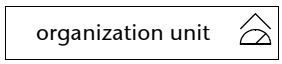
(a) Hosting of an agent



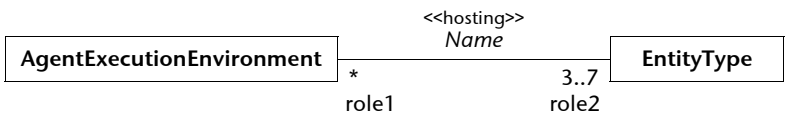
(c) Hosting of an environment

(b) Hosting of a resource

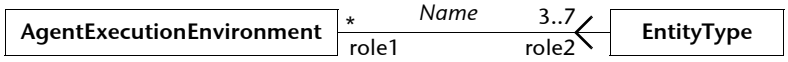
(d) Hosting of an organization unit



HostingAssociation (p. 54)

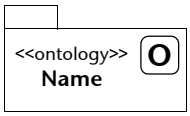


Presentation option:

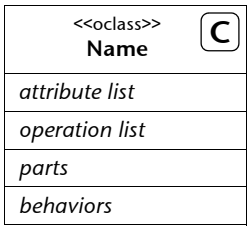


Tab. 11-2 AML notation summary

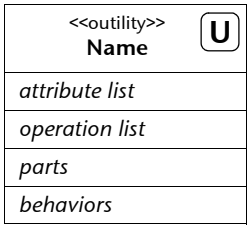
Ontology (p. 56)



OntologyClass (p. 57)



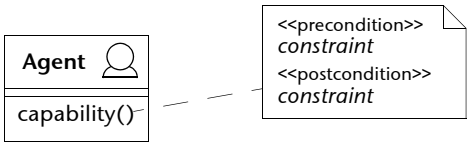
OntologyUtility (p. 58)



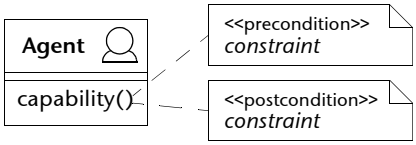
Capability (p. 62)

There is no general notation for Capability.

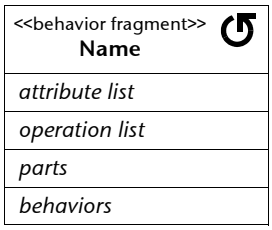
The Capability conditions can be explicitly specified as follows:



Presentation option:

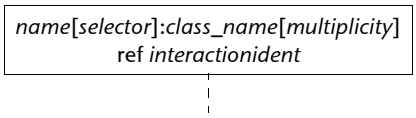


BehaviorFragment (p. 65)

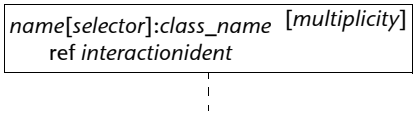


Tab. 11-2 AML notation summary

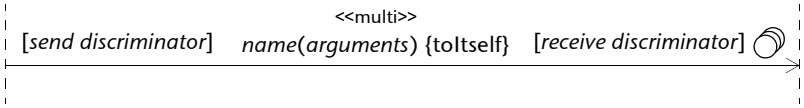
MultiLifeline (p. 70)



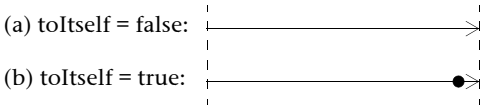
Presentation option:



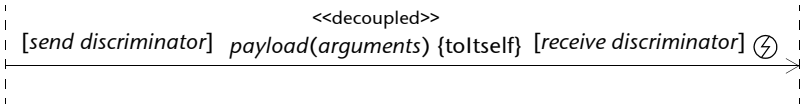
MultiMessage (p. 72)



Presentation option:

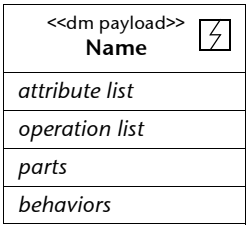


DecoupledMessage (p. 74)



Presentation option as for MultiMessage (p. 72).

DecoupledMessagePayload (p. 76)

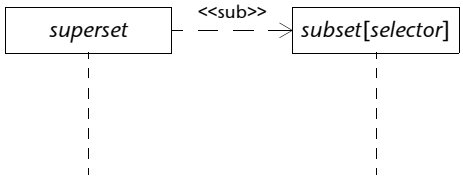


Tab. 11-2 AML notation summary

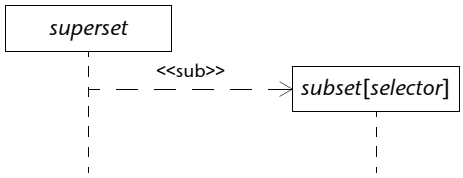
Subset (p. 76)

In Interaction Diagrams:

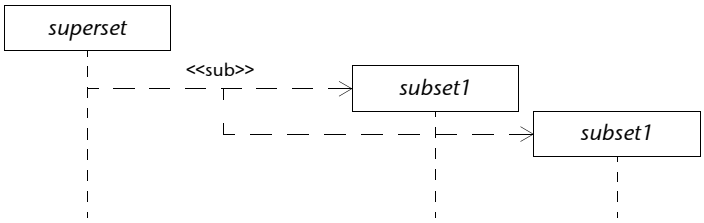
(a) The subset is identified from the beginning of the superset's existence:



(b) The subset is identified during the life time of the superset:

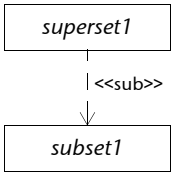


(c) Multiple subsets created at once:

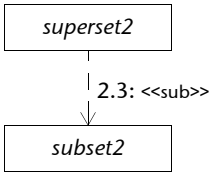


In Communication Diagrams:

(a) *subset1* starts to be identified from when superset1 occurs:



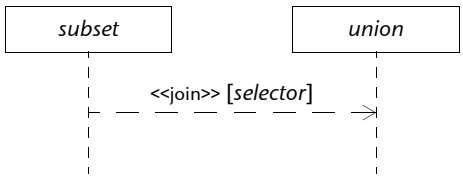
(b) *subset2* is identified when an "event" 2.3 occurs during interaction:



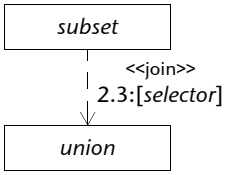
Tab. 11-2 AML notation summary

Join (p. 78)

In Interaction Diagrams:

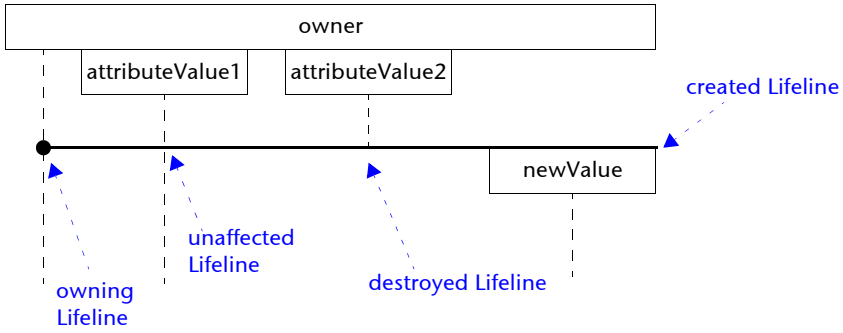


In Communication Diagrams:

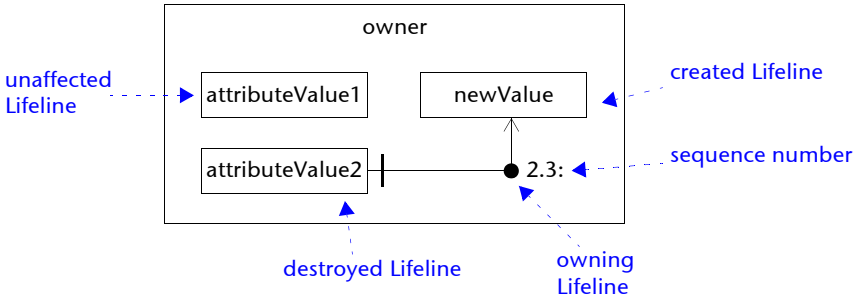


AttributeChange (p. 81)

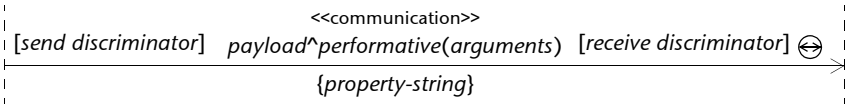
In Interaction Diagrams:



In Communication Diagrams:



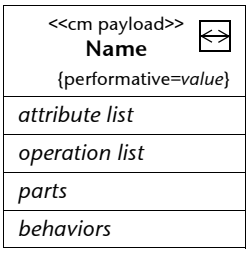
CommunicationMessage (p. 86)



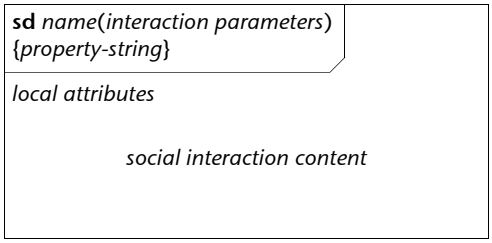
Presentation option as for MultiMessage (p. 72).

Tab. 11-2 AML notation summary

CommunicationMessagePayload (p. 87)

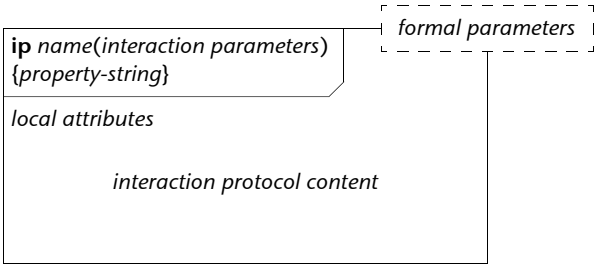


CommunciativeInteraction (p. 89)



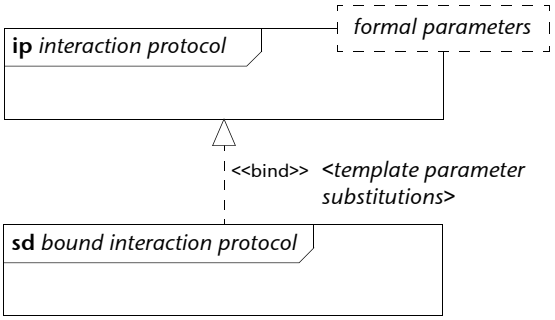
Tab. 11-2 AML notation summary

InteractionProtocol (p. 89)

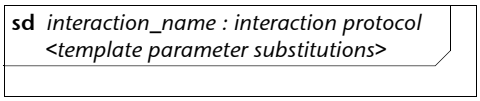


Notation of InteractionProtocol binding:

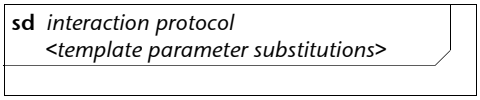
(a) As a TemplateBinding relationship:



(b) As a named bound CommunicativeInteraction:

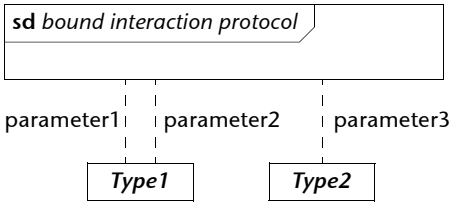


(c) As an anonymous bound CommunicativeInteraction:

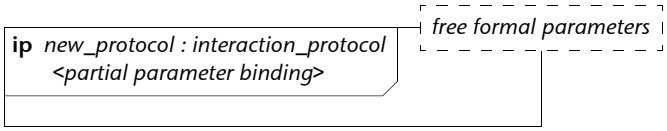


Presentation options:

(a) Binding of formal parameters:

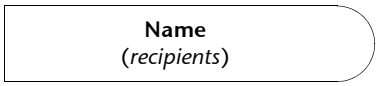


(b) Free formal parameters of a partially bound InteractionProtocol:

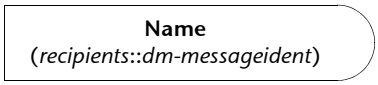


Tab. 11-2 AML notation summary

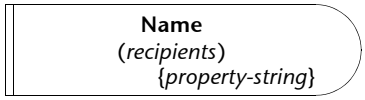
SendDecoupledMessageAction (p. 94)



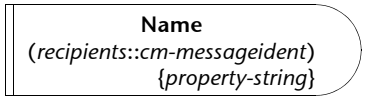
Presentation options:



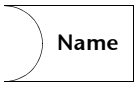
SendCommunicationMessageAction (p. 95)



Presentation options:

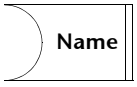


AcceptDecoupledMessageAction (p. 97)



Presentation option:

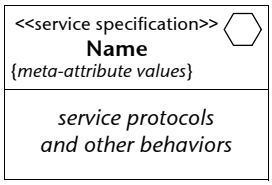
AcceptCommunicationMessageAction (p. 98)



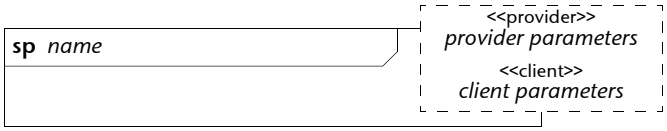
Presentation option:

Tab. 11-2 AML notation summary

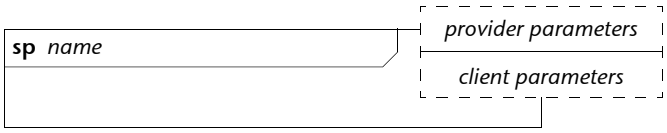
ServiceSpecification (p. 103)



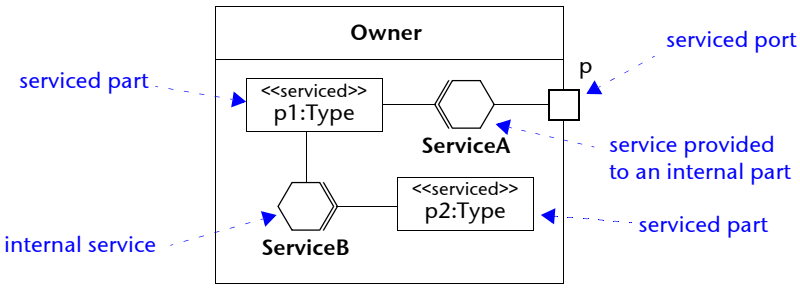
ServiceProtocol (p. 105)



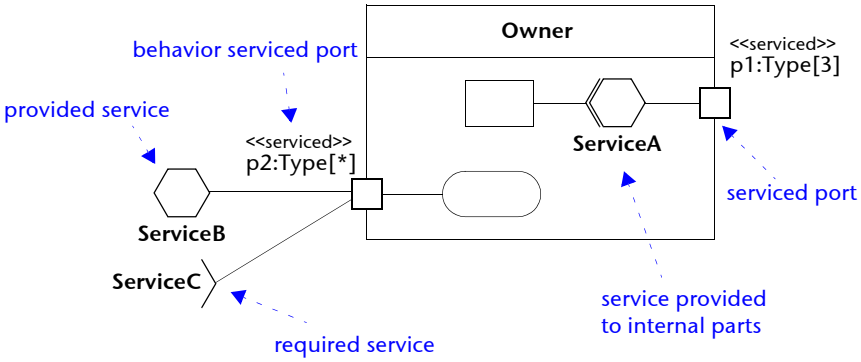
Presentation option:



ServicedProperty (p. 109)

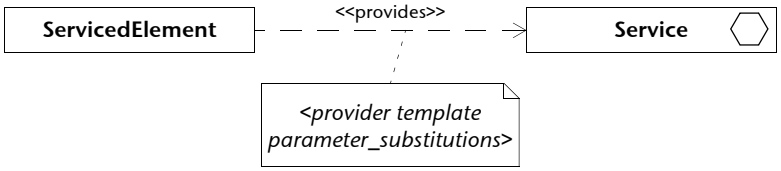


ServicedPort (p. 110)

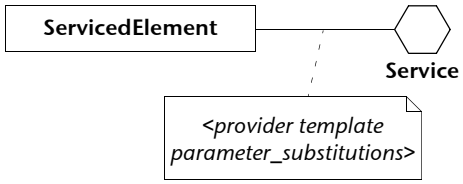


Tab. 11-2 AML notation summary

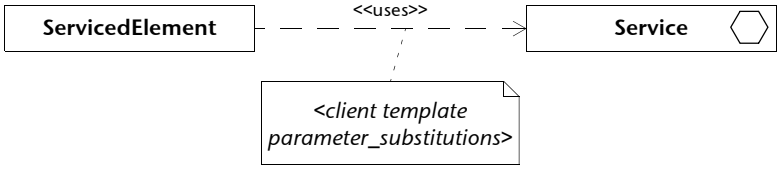
ServiceProvision (p. 112)



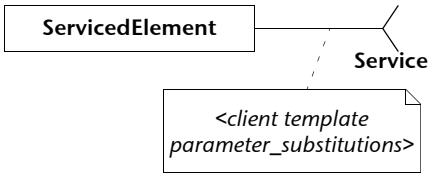
Presentation option:



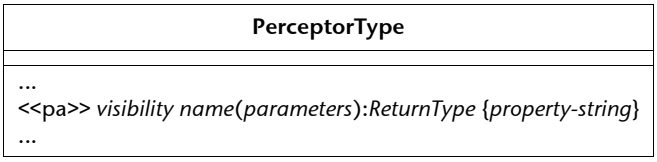
ServiceUsage (p. 114)



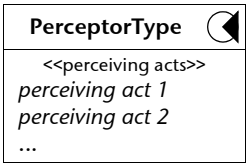
Presentation option:



PerceivingAct (p. 117)

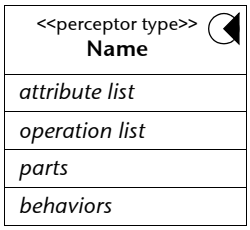


Presentation option:

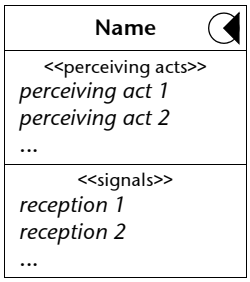


Tab. 11-2 AML notation summary

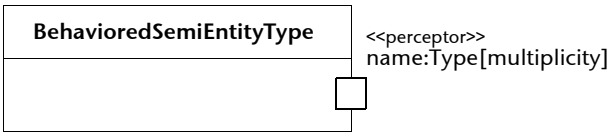
PerceptorType (p. 118)



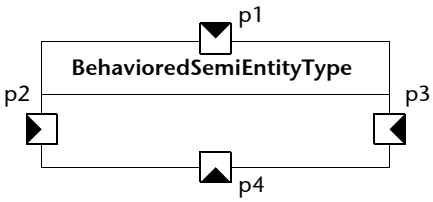
Presentation option:



Perceptor (p. 119)

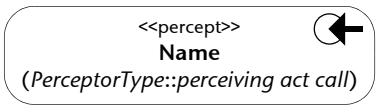


Presentation option:



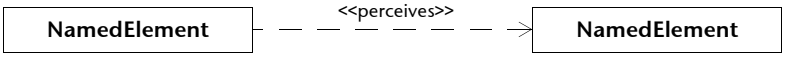
PerceptAction (p. 121)

Presentation option:

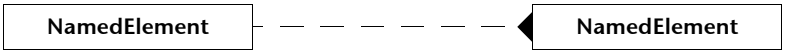


Tab. 11-2 AML notation summary

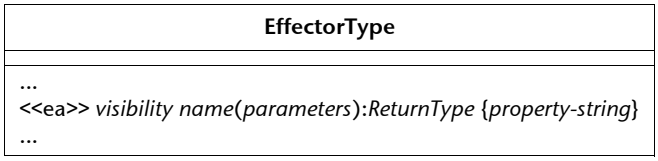
Perceives (p. 123)



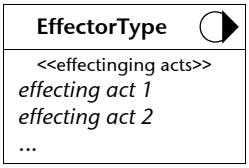
Presentation option:



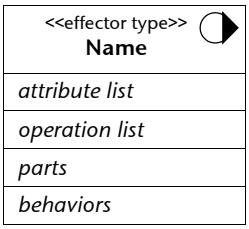
EffectingAct (p. 123)



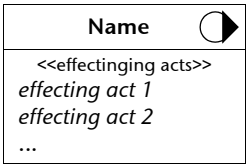
Presentation option:



EffectorType (p. 124)

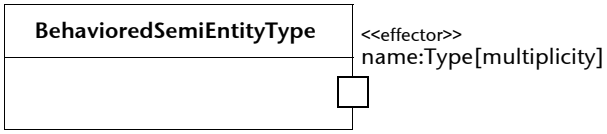


Presentation option:

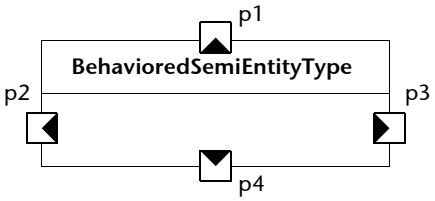


Tab. 11-2 AML notation summary

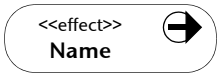
Effector (p. 125)



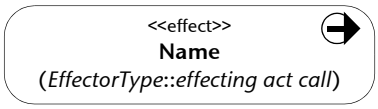
Presentation option:



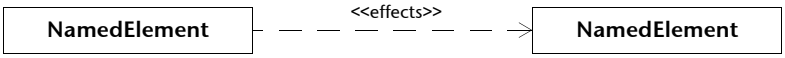
EffectAction (p. 126)



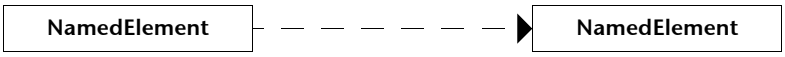
Presentation option:



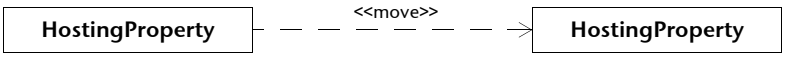
Effects (p. 127)



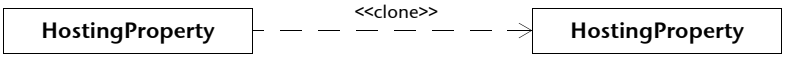
Presentation option:



Move (p. 129)



Clone (p. 130)

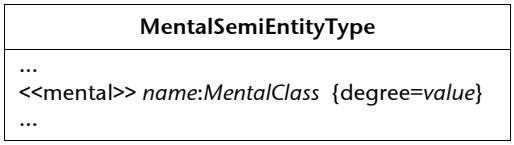


MoveAction (p. 132)

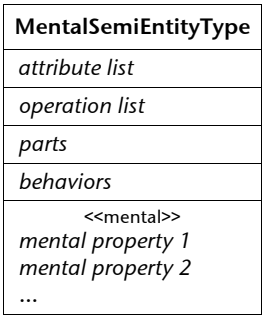
Tab. 11-2 AML notation summary

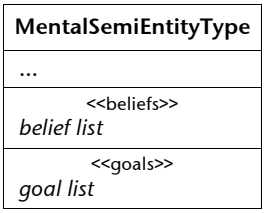
CloneAction (p. 133)

MentalProperty (p. 144)

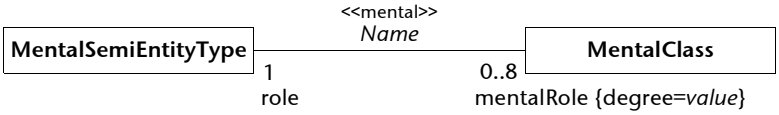


Presentation options:

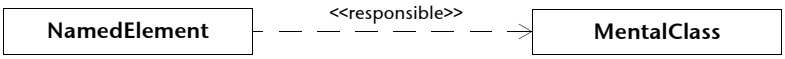




MentalAssociation (p. 147)

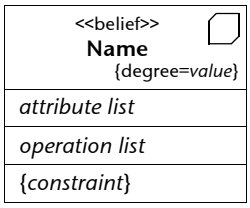


Responsibility (p. 148)

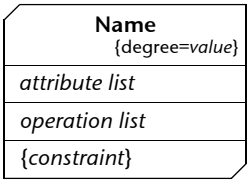


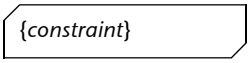
Tab. 11-2 AML notation summary

Belief (p. 149)

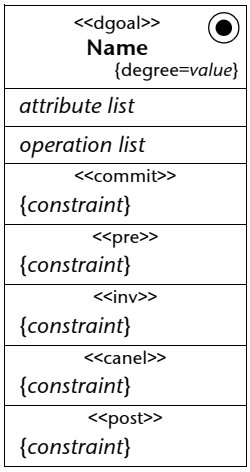


Presentation options:

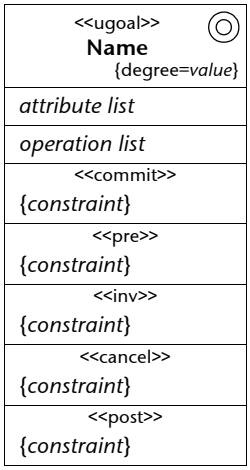




DecidableGoal (p. 152)

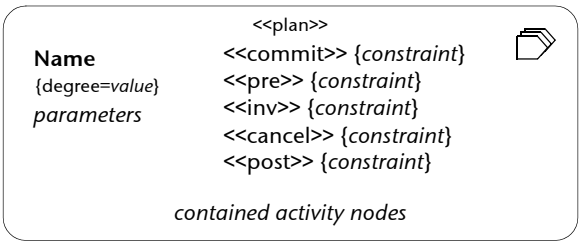


UndecidableGoal (p. 154)



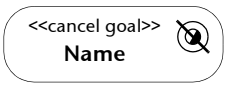
Tab. 11-2 AML notation summary

Plan (p. 156)



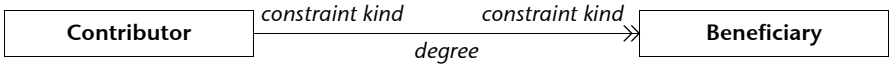
CommitGoalAction (p. 158)

CancelGoalAction (p. 160)

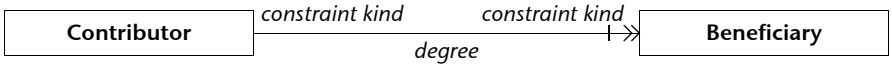


Contribution (p. 162)

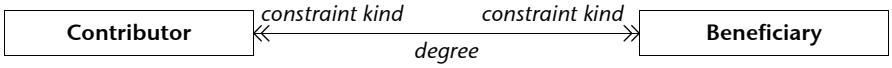
Sufficient:



Necessary:



Equivalent (iff):



Presentation options:

(a) commitCondition: 

(b) preCondition: 

(c) commitPreCondition: 

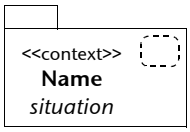
(d) invariant: 

(e) cancelCondition: 

(f) postCondition: 

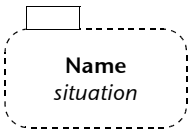
Tab. 11-2 AML notation summary

Context (p. 171)

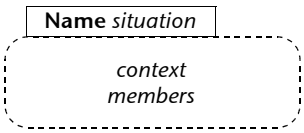


Presentation options:

(a) Hidden members:



(b) Shown members:



Tab. 11-2 AML notation summary

Appendix C Glossary

AcceptCommunicationMessageAction (p. 98)
AcceptCommunicationMessageAction is a specialized AcceptEventAction (from UML) which waits for the reception of a CommunicationMessage (p. 265) that meets conditions specified by the associated trigger (for details see CommunicationMessageTrigger, p. 265).

AcceptDecoupledMessageAction (p. 97)
AcceptDecoupledMessageAction is a specialized AcceptEventAction (from UML) which waits for the reception of a DecoupledMessage (p. 266) that meets conditions specified by the associated trigger (for details see DecoupledMessageTrigger, p. 266).

Agent
An instance of an AgentType (p. 263) which represents a self-contained entity that is capable of autonomous behavior within its environment. It has at least the following features: (1) autonomy (i.e. a control over its own state and behavior based on external or internal stimuli), and (2) ability to interact (i.e. the capability to interact with its environment, including perceptions and effecting actions, speech act based interactions, etc.).

AgentExecutionEnvironment (p. 49)
AgentExecutionEnvironment is a specialized ExecutionEnvironment (from UML) and BehavioredSemiEntityType (p. 264), used to model types of execution environments of multi-agent system. AgentExecutionEnvironment thus provides the physical infrastructure in which MAS entities (p. 267) can run.

AgentType (p. 25)
AgentType is a specialized AutonomousEntityType (p. 263) used to model a type of agents (p. 263).

AttributeChange (p. 81)
AttributeChange is a specialized InteractionFragment (from UML) used to model the change of attribute values (state) of ConnectableElements (from UML) in the context of Interactions (from UML).

Autonomous entity
An instance of an AutonomousEntityType (p. 263).

AutonomousEntityType (p. 24)
AutonomousEntityType is an abstract specialized BehavioralEntityType (p. 264) and MentalSemiEntityType (p. 269), used to model types of self-contained entities that are capable of autonomous behavior in their environment, i.e. entities that have control of their own behavior, and act upon their environment according to the processing of (reasoning on) perceptions of that environment, interactions and/or their mental attitudes. There are no other entities (p. 267) that directly control the behavior of autonomous entities (p. 263).

Behavioral entity
An instance of a BehavioralEntityType (p. 264).

BehavioralEntityType (p. 24)
BehavioralEntityType is an abstract specialized EntitiyType (p. 267) used to represent types of entities (p. 267) which have the features of BehavioredSemiEntityType (p. 264) and SocializedSemiEntityType (p. 273), and can play entity roles (p. 267).

Behaviored semi-entity
An instance of a BehavioredSemiEntityType (p. 264).

BehavioredSemiEntityType (p. 60)
BehavioredSemiEntityType is an abstract specialized Class (from UML) and ServicedElement (p. 272), that serves as a common superclass to all metaclasses which can own capabilities, observe and/or effect their environment, and provide and/or use services (p. 271).

BehaviorFragment (p. 65)
BehaviorFragment is a specialized BehavioredSemiEntityType (p. 264) used to model coherent and reusable fragments of behavior and related structural and behavioral features, and to decompose complex behaviors into simpler and (possibly) concurrently executable fragments.

Belief (p. 149)
Belief is a specialized MentalClass (p. 268) used to model a state of affairs, proposition or other information relevant to the system and its mental model. If an instance of a Belief is held in a slot of a mental semi-entity's MentalProperty (p. 268), it represents the information which the mental semi-entity believes, and which does not need to be objectively true.

CancelGoalAction (p. 160)
CancelGoalAction is a specialized DestroyObjectAction (from UML) used to model de-commitment from goals.

Capability (p. 62)
Capability is an abstract specialized RedefinableElement (from UML) and Namespace (from UML), used to model an abstraction of a behavior in terms of its inputs, outputs, pre-conditions, and post-conditions. Such a common abstraction allows use of the common features of all the concrete subclasses of the Capability metaclass uniformly, and thus reason about and operate on them in a uniform way.

Clone (p. 130)
Clone is a specialized Dependency (from UML) between HostingProperties (p. 268) used to model the possibility of cloning an entity (p. 267).

CloneAction (p. 133)
CloneAction is a specialized MobilityAction (p. 269) used to model the action of cloning.

CommitGoalAction (p. 158)
CommitGoalAction is a specialized CreateObjectAction (from UML) and AddStructuralFeatureValueAction (from UML), used to model the action of commitment to a Goal (p. 267).

CommunicationMessage (p. 86)
CommunicationMessage is a specialized DecoupledMessage (p. 266) and CommunicationSpecifier (p. 265), which is used to model communicative acts of speech act based communication in the context of Interactions.

CommunicationMessagePayload (p. 87)
CommunicationMessagePayload is a specialized Class (from UML) used to model the type of objects transmitted in the form of CommunicationMessages (p. 265).

CommunicationMessageTrigger (p. 100)
CommunicationMessageTrigger is a specialized DecoupledMessageTrigger (p. 266) that represents the event of reception of a CommunicationMessage (p. 265), that satisfies the specified condition.

CommunicationSpecifier (p. 85)
CommunicationSpecifier is an abstract metaclass which defines metaproperties of its concrete subclasses which are used to model different aspects of communicative interactions.

CommunicativeInteraction (p. 89)
CommunicativeInteraction is a specialized Interaction (from UML) and CommunicationSpecifier (p. 265), used to model speech act based communications, i.e. Interactions containing CommunicationMessages (p. 265).

ConstrainedMentalClass (p. 140)
ConstrainedMentalClass is an abstract specialized MentalClass (p. 268) which allows its concrete subclasses to specify MentalConstraints (p. 268).

Context (p. 171)
Context is a specialized Package (from UML) used to logically structure models according to particular situations.

Contribution (p. 162)
Contribution is a specialized MentalRelationship (p. 269) and DirectedRelationship (from UML) used to model logical relationships between MentalStates (p. 269) and their MentalConstraints (p. 268).

ContributionKind (p. 169)
ContributionKind is an enumeration which specifies possible kinds of Contributions (p. 265).

CreateRoleAction (p. 45)
CreateRoleAction is a specialized CreateObjectAction (from UML) and AddStructuralFeatureValueAction (from UML), used to model the action of creating and starting to play an entity role (p. 267) by a behavioral entity (p. 263).

DecidableGoal (p. 152)
DecidableGoal is a specialized concrete Goal (p. 267) used to model goals for which there are clear-cut criteria according to which the goalholder (p. 267) can decide whether the DecidableGoal (particularly its postCondition) has been achieved or not.

DecoupledMessage (p. 74)
DecoupledMessage is a specialized MultiMessage (p. 269) which is used to model a specific kind of communication within an Interaction (from UML), particularly the asynchronous sending and receiving of a DecoupledMessagePayload (p. 266) instance without explicit specification of the behavior invoked on the side of the receiver. The decision of which behavior should be invoked when the DecoupledMessage (p. 266) is received is up to the receiver (for details see DecoupledMessageTrigger, p. 266 and AcceptDecoupledMessageAction, p. 263).

DecoupledMessagePayload (p. 76)
DecoupledMessagePayload is a specialized Class (from UML) used to model the type of objects transmitted in the form of DecoupledMessages (p. 266).

DecoupledMessageTrigger (p. 99)
DecoupledMessageTrigger is a specialized Trigger (from UML) that represents the event of reception of a DecoupledMessage (p. 266), that satisfies the specified condition.

DisposeRoleAction (p. 47)
DisposeRoleAction is a specialized DestroyObjectAction (from UML) used to model the action of stopping to play an entity role (p. 267) by a behavioral entity (p. 263).

EffectAction (p. 126)
EffectAction is a specialized CallOperationAction (from UML) which can call EffectingActs (p. 266). Thus, an EffectAction can transmit an operation call request to an EffectingAct, which causes the invocation of the associated behavior.

EffectingAct (p. 123)
EffectingAct is a specialized Operation (from UML) which is owned by a EffectorType (p. 266) and thus can be used to specify what effecting acts the owning EffectorType, or an Effector (p. 266) of that EffectorType, can perform.

Effector (p. 125)
Effector is a specialized ServicedPort (p. 272) used to model the capability of its owner (a BehavioredSemiEntityType, p. 264) to bring about an effect on others, i.e. to directly manipulate with (or modify a state of) some other objects. What effects an Effector is capable of is specified by its type, i.e. EffectorType (p. 266).

EffectorType (p. 124)
EffectorType is a specialized BehavioredSemiEntityType (p. 264) used to model type of Effectors (p. 266), in terms of owned EffectingActs (p. 266).

Effects (p. 127)
Effects is a specialized Dependency (from UML) used to model which elements can effect others.

Entity
An instance of an EntityType (p. 267), i.e. an object, which can exist in the system independently of other objects, e.g. an agent (p. 263), resource (p. 271), environment (p. 267).

Entity role
An instance of an EntityRoleType (p. 267).

EntityRoleType (p. 39)
EntityRoleType is a specialized BehavioredSemiEntityType (p. 264), MentalSemiEntityType (p. 269), and SocializedSemiEntityType (p. 273), used to represent a coherent set of features, behaviors, participation in interactions, and services (p. 271) offered or required by BehavioralEntityTypes (p. 264) in a particular context (e.g. interaction or social).

EntityType (p. 23)
EntityType is an abstract specialized Type (from UML). It is a superclass to all AML modeling elements which represent types of entities (p. 267) of an MAS.

Environment
A logical or physical surroundings of entities which provide conditions under which the entities exist and function. From the point of view of the (multi-agent) system modeled, two categories of environments can be recognized: (1) ***System internal environment***, which is a part of the system modeled, and (2) ***system external environment***, which is outside the system modeled and forms the boundaries onto that system.

EnvironmentType (p. 28)
EnvironmentType is a specialized AutonomousEntityType (p. 263) used to model types of environments (p. 267). EnvironmentType thus can be used to model particular aspects of the world which entities inhabit, its structure and behavior. It can contain the space and all the other objects in the entity surroundings, but also those principles and processes (i.e. laws, rules, constraints, policies, services, roles, resources, etc.) which together constitute the circumstances under which entities act.

Goal (p. 151)
Goal is an abstract specialized ConstrainedMentalClass (p. 265) used to model goals, i.e. conditions or states of affairs, with which the main concern is their achievement or maintenance. The Goals can thus be used to represent objectives, needs, motivations, desires, etc.

Goal-holder
Goal-holder is the stakeholder or mental semi-entity (p. 268) which has control over a goal. Stakeholder is an individual who is affected by the system, or by the system modeling process.

Hosting link
An instance of a HostingAssociation (p. 267).

HostingAssociation (p. 54)
HostingAssociation is a specialized Association (from UML) used to specify HostingProperty (p. 268) in the form of an association end.

HostingKind (p. 54)
HostingKind is an enumeration which specifies possible hosting relationships of EntityTypes (p. 267) to AgentExecutionEnvironments (p. 263) (e.g. resident and visitor).

HostingProperty (p. 51)
HostingProperty is a specialized ServicedProperty (p. 272) used to specify what EntityTypes (p. 267) can be hosted by what AgentExecutionEnvironments (p. 263).

InteractionProtocol (p. 89)
InteractionProtocol is a parametrized CommunicativeInteraction (p. 265) template used to model reusable templates of CommunicativeInteractions.

Join (p. 78)
Join is a specialized Dependency (from UML) used to specify joining of instances represented by one Lifeline (from UML) with a set of instances represented by another Lifeline.

Mental link
An instance of a MentalAssociation (p. 268).

Mental semi-entity
An instance of a MentalSemiEntityType (p. 269).

MentalAssociation (p. 147)
MentalAssociation is a specialized Association (from UML) between a MentalSemiEntityType (p. 269) and a MentalClass (p. 268) used to specify a MentalProperty (p. 268) of the MentalSemiEntityType in the form of an association end.

MentalClass (p. 140)
MentalClass is an abstract specialized Class (from UML) and MentalState (p. 269) serving as a common superclass to all the metaclasses which can be used to specify mental attitudes of MentalSemiEntityTypes (p. 269).

MentalConstraint (p. 141)
MentalConstraint is a specialized Constraint (from UML) and RedefinableElement (from UML), used to specify properties of ConstrainedMentalClasses (p. 265) which can be used within mental (reasoning) processes of owning MentalSemiEntityTypes (p. 269). For further details see also MentalConstraintKind (p. 268).

MentalConstraintKind (p. 142)
MentalConstraintKind is an enumeration which specifies kinds of MentalConstraints (p. 268), as well as kinds of constraints specified for contributor and beneficiary in the Contribution (p. 265) relationship.

MentalProperty (p. 144)
MentalProperty is a specialized Property (from UML) used to specify that instances of its owner (i.e. mental semi-entities, p. 268) have control over instances of the MentalClasses (p. 268) of its type, e.g. can decide whether to believe or not (and to what extent) in a Belief (p. 264), or whether and when to commit to a Goal (p. 267).

MentalRelationship (p. 143)
MentalRelationship is an abstract specialized MentalState (p. 269), a superclass to all metaclasses defining the relationships between MentalStates.

MentalSemiEntityType (p. 143)
MentalSemiEntityType is a specialized abstract Class (from UML), a superclass to all metaclasses which can own MentalProperties (p. 268), i.e. AutonomousEntityType (p. 263) and EntityRoleType (p. 267).

MentalState (p. 139)
MentalState is an abstract specialized NamedElement (from UML) serving as a common superclass to all metaclasses which can be used for: (1) modeling mental attitudes of MentalSemiEntityTypes (p. 269), which represent their informational, motivational and deliberative states, and (2) support for the human mental process of requirements specification and analysis of complex problems/systems, e.g. expressing intentionality in use case models, goal-based requirements modeling, or problem decomposition.

MobilityAction (p. 131)
MobilityAction is an abstract specialized AddStructuralFeatureValueAction (from UML) used to model mobility actions of entities (p. 267), i.e. actions that cause movement or cloning of an entity from one AgentExecutionEnvironment (p. 263) to another one.

Move (p. 129)
Move is a specialized Dependency (from UML) between two used to model the possibility of moving an entity (p. 267) between instances of AgentExecutionEnvironments (p. 263).

MoveAction (p. 132)
MoveAction is a specialized MobilityAction (p. 269) used to model an action of moving an entity (p. 267) between instances of AgentExecutionEnvironments (p. 263).

MultiLifeline (p. 70)
MultiLifeline is a specialized Lifeline (from UML) and MultiplicityElement (from UML), used to represent a multivalued ConnectableElement (from UML) (i.e. ConnectableElement with multiplicity > 1) participating in an Interaction (from UML).

MultiMessage (p. 72)
MultiMessage is a specialized Message (from UML) which is used to model a particular communication between MultiLifelines (p. 269) of an Interaction (from UML).

Ontology (p. 56)
Ontology is a specialized Package (from UML) used to specify a single ontology.

OntologyClass (p. 57)
OntologyClass is a specialized Class (from UML) used to represent an ontology class (called also ontology concept or frame).

OntologyUtility (p. 58)
OntologyUtility is a specialized Class (from UML) used to cluster global *ontology constants*, *ontology variables*, and *ontology functions/actions/predicates* modeled as owned features. The features of an OntologyUtility can be used by (referred to by) other elements within the owning and importing Ontologies (p. 269).

Organization unit
An instance of an OrganizationUnitType (p. 270) which represents a social environment or its part. From an ***external perspective***, organization units represent coherent autonomous entities (p. 263) which can have goals, perform behavior, interact with their environment, offer services, play roles, etc. Properties and behavior of organization units are both: (1) emergent properties and behavior of all their constituents, their mutual relationships, observations and interactions, and (2) the features and behavior of organization units themselves. From an ***internal perspective***, organization units are types of environment that specify the social arrangements of entities (p. 267) in terms of structures, interactions, roles, constraints, norms, etc.

OrganizationUnitType (p. 31)
OrganizationUnitType is a specialized EnvironmentType (p. 267) used to model types of organization units (p. 270).

Perceives (p. 123)
Perceives is a specialized Dependency (from UML) used to model which elements can observe others.

PerceivingAct (p. 117)
PerceivingAct is a specialized Operation (from UML) which is owned by a PerceptorType (p. 270) and thus can be used to specify what perceptions the owning PerceptorType, or a Perceptor (p. 270) of that PerceptorType, can perform.

PerceptAction (p. 121)
PerceptAction is a specialized CallOperationAction (from UML) which can call PerceivingActs (p. 270). As such, PerceptAction can transmit an operation call request to a PerceivingAct, what causes the invocation of the associated behavior.

Perceptor (p. 119)
Perceptor is a specialized ServicedPort (p. 272) used model capability of its owner (a BehavioredSemiEntityType, p. 264) to observe, i.e. perceive a state of and/or to receive a signal from observed objects. What observations a Perceptor is capable of is specified by its type, i.e. PerceptorType (p. 270).

PerceptorType (p. 118)
PerceptorType is a specialized BehavioredSemiEntityType (p. 264) used to model the type of Perceptors (p. 270), in terms of owned (1) Receptions (from UML) and (2) PerceivingActs (p. 270).

Plan (p. 156)
Plan is a specialized ConstrainedMentalClass (p. 265) and Activity (from UML), used to model capabilities of MentalSemiEntityTypes (p. 269)

which represents either: (1) predefined plans, i.e. kinds of activities a mental semi-entity's (p. 268) reasoning mechanism can manipulate in order to achieve Goals (p. 267), or (2) fragments of behavior from which the plans can be composed (called also *plan fragments*).

Play link
An instance of a PlayAssociation (p. 271).

PlayAssociation (p. 43)
PlayAssociation is a specialized Association (from UML) used to specify RoleProperty (p. 271) in the form of an association end.

Resource
An instance of a ResourceType (p. 271), i.e. a physical or an informational entity, with which the main concern is its availability and usage (e.g. quantity, access rights, conditions of consumption).

ResourceType (p. 27)
ResourceType is a specialized BehavioralEntityType (p. 264) used to model types of resources (p. 271) contained within the system.

Responsibility (p. 148)
Responsibility is a specialized Realization (from UML) used to model a relation between MentalClasses (p. 268) and NamedElements (from UML) that are obligated to accomplish (or to contribute to the accomplishment of) those MentalClass (e.g. modification of Beliefs, p. 264, or achievement or maintenance of Goals, p. 267, or realization of Plans, p. 270).

RoleProperty (p. 41)
RoleProperty is a specialized Property (from UML) used to specify that an instance of its owner, a BehavioralEntityType (p. 264), can play one or several entity roles (p. 267) of the specified EntityRoleType (p. 267).

Semi-entity
Semi-entity is a modeling concept that defines certain aspects and features of entities (p. 267), but itself does not represent an entity.

SendCommunicationMessageAction (p. 95)
SendCommunicationMessageAction is a specialized SendDecoupledMessageAction (p. 271), which allows to specify the values of the CommunicationSpecifier's (p. 265) meta-attributes.

SendDecoupledMessageAction (p. 94)
SendDecoupledMessageAction is a specialized SendObjectAction (from UML) used to model the action of sending of DecoupledMessagePayloads (p. 266) instances in the form of a DecoupledMessage (p. 266).

Service
Service is a coherent block of functionality provided by a behaviored semi-entity (p. 264), called service provider (p. 272), that can be accessed by other behaviored semi-entities (which can be either external or internal parts of the service provider), called service clients (p. 271).

Service client
Service client is a behaviored semi-entity (p. 264) that uses a service (p. 271).

Service provider
Service provider is a behaviored semi-entity (p. 264) that provides a service (p. 271).

ServicedElement (p. 108)
ServicedElement is an abstract specialized NamedElement (from UML) used to serve as a common superclass to all the metaclasses that can provide or use services (p. 271).

ServicedPort (p. 110)
ServicedPort is a specialized Port (from UML) and ServicedElement (p. 272) that specifies a distinct interaction point between the owning BehavioredSemiEntityType (p. 264) and other ServicedElements in the model. The nature of the interactions that may occur over a ServicedPort (p. 272) can, in addition to required and provided interfaces, be specified also in terms of required and provided services (p. 271), particularly by associated provided and/or required ServiceSpecifications (p. 272).

ServicedProperty (p. 109)
ServicedProperty is a specialized Property (from UML) and ServicedElement (p. 272), used to model attributes that can provide or use services (p. 271). It determines what services are provided and used by the behaviored semi entities (p. 264) when occur as attribute values of some objects.

ServiceProtocol (p. 105)
ServiceProtocol is a specialized InteractionProtocol (p. 268) used to specify how the functionality of a service (p. 271) can be accessed.

ServiceProvision (p. 112)
ServiceProvision is a specialized Realization dependency (from UML) between a ServiceSpecification (p. 272) and a ServicedElement (p. 272), used to specify that the ServicedElement provides the service (p. 271) specified by the related ServiceSpecification.

ServiceSpecification (p. 103)
ServiceSpecification is a specialized BehavioredClassifier (from UML) and CommunicationSpecifier (p. 265), used to specify services (p. 271).

ServiceUsage (p. 114)
ServiceUsage is a specialized Usage dependency (from UML) between a ServiceSpecification (p. 272) and a ServicedElement (p. 272), used to specify that the ServicedElement uses or requires (can request) the service (p. 271) specified by the related ServiceSpecification.

Social link
An instance of a SocialAssociation (p. 272).

SocialAssociation (p. 38)
SocialAssociation is a specialized Association (from UML) used to model social relationships that can occur between SocializedSemiEntityTypes (p. 273).

Socialized semi-entity
An instance of a SocializedSemiEntityType (p. 273).

SocializedSemiEntityType (p. 33)

SocializedSemiEntityType is an abstract specialized Class (from UML), a superclass to all metaclasses which can participate in SocialAssociatons (p. 272) and can own SocialProperties (p. 273).

SocialProperty (p. 35)

SocialProperty is a specialized ServicedProperty (p. 272) used to specify social relationships that can or must occur between instances of its type and: (1) instances of its owning class (when the SocialProperty is an attribute of a Class), or (2) instances of the associated class (when the SocialProperty is a member end of an Association (from UML)).

SocialRoleKind (p. 38)

SocialRoleKind is an enumeration which specifies allowed values for the socialRole meta-attribute of the SocialProperty (p. 273).

Subset (p. 76)

Subset is a specialized Dependency (from UML) used to specify that instances represented by one Lifeline (from UML) are a subset of instances represented by another Lifeline.

UndecidableGoal (p. 154)

UndecidableGoal is a specialized concrete Goal (p. 267) used to model goals for which there are no clear-cut criteria according to which the goal-holder (p. 267) can decide whether the post-condition of the UndecidableGoal is achieved or not.