FORMAL SPECIFICATION OF AML
MASTER’S THESIS

JÁN DANČ

Advisor:
Mgr. Radovan Červenka, PhD.
Bratislava, 2008
Formal Specification of AML

Master’s Thesis

Ján Danč

Thesis advisor: Mgr. Radovan Červenka, PhD.

Comenius University
Faculty of Mathematics, Physics and Informatics
Department of Computer Science

Branch of study: Informatics

Bratislava, May 2008
I hereby declare that I wrote this thesis by myself, only with the help of the referenced literature, under the careful supervision of my thesis advisor.
Acknowledgement

I would like to thank to my advisor Mgr. R. Červenka, Phd. for his invaluable guidance and many useful suggestions during my work on this thesis.

I would also like to express my gratitude to all those who gave me the possibility to complete this thesis.

Special thanks goes to Dr. Soon-Kyeong Kim from the University of Queensland in Australia, who patiently answered my questions and helped me direct my AML transformation attempts in their early stages.
Abstract

In our work we present a formal specification of Agent Modeling Language (AML), which was defined in PhD thesis wrote by R. Červenka. It provides precise and unambiguous description of AML, which can help in better understanding of the language, opening the door to its wider applicability and further improvements.

We also show how to formally specify an abstract multi-agent system (MAS) by means of the concepts of AML. This part demonstrates and proves well-formedness of the used concepts and provides a basis for further research in the area of MAS theories and formal specification of agent-based systems.

Keywords: AML, Agent Modeling Language, MAS, Multi-Agent Systems, Object-Z, OZ, formal specification.
Preface

“Of course, there is no fool-proof methodology or magic formula that will ensure a good, efficient, or even feasible design. For that, the designer needs experience, insight, flair, judgement, invention. Formal methods can only stimulate, guide, and discipline our human inspiration, clarify design alternatives, assist in exploring their consequences, formalize and communicate design decisions, and help to ensure that they are correctly carried out.”

C.A.R. Hoare, 1988

Formal methods are becoming more accepted in both academia and industry as one possible way in which to help improve the quality of both software and hardware systems. It should be remembered however that they are not a panacea, but rather one more weapon in the armory against making design mistakes. Thus we should not expect too much from formal methods, but rather use them to advantage where appropriate. [19]

The work on which this thesis is based involved, mainly, the PhD thesis wrote by R. Červenka [2]. Our thesis started by gathering some background informations about AML, Z and Object-Z notation. Our first goal was to transform the AML specification presented in UML diagrams to Object-Z specification. For this purpose an automatic transformation engine would be an ideal solution. Unfortunately, due to organization problems our effort has not been attended with success, and therefore, we were forced to make the transformation manually using a formal mapping. Secondly, in order to properly comprehend the content of AML, it was necessary to understand its underlying concepts. A formal specification of a model of an abstract multi-agent system was provided to describe them. R. Červenka in [2] writes: “The intention is not to provide a comprehensive metamodel for all aspects and details of a MAS (e.g. detailed architectural design, system dynamics, operational semantics, etc.), but rather to explain the concepts that were used as the underlying principles of AML, and influenced the design of comprised modeling constructs.”

We hope that our work can serve as a basis for more specific investigations in a multi-agent theory.

Bratislava, May 2008

Ján Danč
# Contents

1 Introduction ................................. 1  
1.1 Motivation and Goals of Thesis ................. 1  
1.2 Structure of Thesis ......................... 2  

2 Introduction to AML ......................... 3  

3 Object-Z in Shortcut ......................... 4  

I Solution Summary .............................. 6  

4 Transformation of the AML Metamodel ........... 7  

5 Object-Z and the Formal Mapping ............... 9  
5.1 Formal Mapping Between UML Models and Object-Z Specifications .......... 9  
5.2 Usage of the Formal Mapping .................. 13  
5.2.1 Mapping Classes ............................. 13  
5.2.2 Mapping Attributes ........................... 14  
5.2.3 Mapping Operations ........................... 14  
5.2.4 Mapping Associations ........................ 14  
5.2.5 Mapping Aggregation and Composition ........ 14  
5.2.6 Mapping Association Classes ................. 15  
5.2.7 Mapping Generalization ....................... 16  
5.3 Additional Functions ......................... 16  
5.4 Example of Mapping ............................. 17  

II Formal Specification of AML .................. 19  

6 Overview of used UML classes ................... 20  

7 Organization of the AML Specification ........... 21 

8 Architecture

8.1 Entities ............................................................... 23
  8.1.1 EntityType .......................................................... 23
  8.1.2 BehavioralEntityType ................................................. 24
  8.1.3 AutonomousEntityType ............................................. 24

8.2 Agents ..................................................................... 25
  8.2.1 AgentType .............................................................. 25

8.3 Resources ............................................................... 25
  8.3.1 ResourceType ............................................................ 25

8.4 Environments .......................................................... 26
  8.4.1 EnvironmentType ....................................................... 26

8.5 Social Aspects .......................................................... 26
  8.5.1 OrganizationUnitType .................................................. 27
  8.5.2 SocializedSemiEntityType ........................................... 27
  8.5.3 SocialProperty .......................................................... 28
  8.5.4 SocialRoleKind ........................................................ 29
  8.5.5 SocialAssociation ...................................................... 30
  8.5.6 EntityRoleType .......................................................... 30
  8.5.7 RoleProperty ............................................................ 31
  8.5.8 PlayAssociation ........................................................ 31
  8.5.9 CreateRoleAction ....................................................... 32
  8.5.10 DisposeRoleAction .................................................... 33

8.6 MAS Deployment ........................................................ 34
  8.6.1 AgentExecutionEnvironment ....................................... 34
  8.6.2 HostingProperty ......................................................... 35
  8.6.3 HostingKind ............................................................. 36
  8.6.4 HostingAssociation ..................................................... 37

9 Behaviors ................................................................. 38

9.1 Basic Behaviors ........................................................ 38
  9.1.1 BehavioredSemiEntityType .......................................... 38
  9.1.2 Capability .............................................................. 40

9.2 Behavior Decomposition .............................................. 42
  9.2.1 BehaviorFragment ...................................................... 42
12 Model Management 85
  12.1 Contexts ................................................................. 85
    12.1.1 Context ........................................................... 85

13 UML Extension for AML 86
  13.1 Extended Actor .......................................................... 86
  13.2 Extended BehavioralFeature ........................................ 87
  13.3 Extended Behavior ...................................................... 87

III Abstract Multi-Agent Framework 88

14 Concepts of AML 89
  14.1 Operation Templates .................................................. 89
  14.2 Multi-Agent System .................................................... 90
    14.2.1 MAS ................................................................. 90
  14.3 MAS Semi-entities and Entities ..................................... 91
    14.3.1 StructuralSemiEntity ............................................. 91
    14.3.2 DeployableSemiEntity ........................................... 93
    14.3.3 CapableSemiEntity ............................................... 93
    14.3.4 BehavioredSemiEntity .......................................... 95
    14.3.5 SocializedSemiEntity ........................................... 95
    14.3.6 MentalSemiEntity ................................................ 95
    14.3.7 Object ............................................................. 95
    14.3.8 Entity ............................................................. 96
    14.3.9 BehavioralEntity ................................................ 97
    14.3.10 Resource ........................................................ 97
    14.3.11 AutonomousEntity .............................................. 97
    14.3.12 Agent ............................................................ 98
    14.3.13 Environment .................................................... 98
  14.4 Social Aspects ........................................................ 98
    14.4.1 SocialRelationshipKind ......................................... 99
    14.4.2 SocialRelationship ................................................ 99
    14.4.3 SocializedSemiEntity ........................................... 99
    14.4.4 BehavioralEntity ................................................ 100
    14.4.5 EntityRole ....................................................... 101
    14.4.6 OrganizationUnit ................................................ 101
14.5 MAS Deployment and Mobility ......................................................... 102
  14.5.1 HostingKind ................................................................. 102
  14.5.2 Hosting ................................................................. 102
  14.5.3 AgentExecutionEnvironment ........................................... 103
14.6 Communicative Interactions ...................................................... 104
  14.6.1 Message .................................................................. 104
  14.6.2 MultiMessage ............................................................. 105
  14.6.3 DecoupledMessage ....................................................... 105
  14.6.4 SendMessageCapability .................................................. 106
  14.6.5 ReceiveMessageCapability .............................................. 106
14.7 Behaviors ............................................................................. 106
  14.7.1 Perceptor .................................................................. 106
  14.7.2 Effector .................................................................... 107
  14.7.3 Service ...................................................................... 108
  14.7.4 BehavioredSemiEntity ..................................................... 108
  14.7.5 BehaviorFragment ......................................................... 110
14.8 Mental Aspects ...................................................................... 110
  14.8.1 ContributionKind .......................................................... 110
  14.8.2 Contribution ............................................................... 110
  14.8.3 MentalAttitude .............................................................. 111
  14.8.4 MentalState ................................................................. 112
  14.8.5 MentalSemiEntity ........................................................ 112
  14.8.6 Belief ....................................................................... 113
  14.8.7 Goals ...................................................................... 113
  14.8.8 Plan ........................................................................ 114
14.9 Autonomy Aspects ................................................................... 114
  14.9.1 BDIAutonomousEntity ..................................................... 114

IV Summary of Achievements .......................................................... 117
15 Conclusions and Future Works ..................................................... 118

Bibliography ................................................................................. 119
List of Figures

3.1 Basic structure of a class in Object-Z .................................. 5
4.1 Generalized Reverse-MDA ................................................. 8
5.1 An UML class diagram for persons and bank accounts ............. 13
5.2 Object-Z class skeleton for class Account ............................... 15
5.3 An UML class diagram for persons and bank accounts ............. 15
5.4 MentalProperty class in AML Metamodel ............................... 18
5.5 MentalProperty class schema without mapped OCL constraints .... 18
7.1 Overall package structure of the AML metamodel ................. 22
Chapter 1

Introduction

“The main approach advocated for making software more reliable is the use of formal, or mathematical, methods of software specification, verification and refinement.”

— Graeme Paul Smith
An Object-Oriented Approach to Formal Specification, 1992

The Agent Modeling Language (AML) [3] is a semi-formal visual modeling language for specifying, modeling and documenting systems that incorporate concepts drawn from multi-agent systems (MAS) theory. It is specified as an extension to UML 2.0, is a consistent set of modeling constructs designed to capture the aspects of multi-agent systems. The ultimate objective for AML is to provide a means for software engineers to incorporate aspects of multi-agent system engineering into their analysis and design processes.

Unified Modeling Language (UML) [11] has been developed as a standard language for object-oriented designs. Through its graphical and intuitive diagrams, software analysis and design process become easy. However, this graphical notation lacks precisely defined semantics. It is difficult to determine whether the design is consistent, unambiguous and complete.

This thesis presents a way of formalizing the AML metamodel. It uses a formal transformation mapping between UML models and Object-Z specifications. With this approach, the semantics of AML are directly expressed in formal specification language Object-Z.

We restrict the scope of our work mainly on describing the principles from theory of agent-based systems and MAS. It is assumed that the reader has the necessary background to understand the presented work.

1.1 Motivation and Goals of Thesis

The most significant motivation driving the development of AML was the extant need for a ready-to-use, comprehensive, versatile and highly expressive modeling language suitable for the development of commercial software solutions based on multi-agent technologies.

To qualify this more precisely, AML was intended to be a language that: (1) is built on proved technical foundations, (2) integrates best practices from agent-oriented software engineering (AOSE) and object-oriented software engineering (OOSE) domains, (3) is well specified and documented, (4) is internally consistent from the conceptual, semantic and syntactic perspectives, (5) is versatile and easy to extend, (6) is independent of any
particular theory, software development process or implementation environment, and (7) is supported by Computer-Aided Software Engineering (CASE) tools. [4]

Motivation behind this thesis is to provide a better insight into multi-agent theory. With a formal specification of AML we will gain access to a deeper understanding of this agent modeling language and also be able to formally describe a multi-agent system on concepts originated in AML.

Formal specification is the first step in the formal development of a software system. It is followed by a series of steps involving verification and refinement which lead to an eventual implementation. The primary role of the formal specification is to provide a precise and unambiguous description of the system as a basis for these subsequent steps. [18]

More about Object-Z and the formal specification can be found in chapters 4 Object-Z in Shortcut (p. 4) and 5 Object-Z and the Formal Mapping (p. 9).

The goals of this thesis can be summarized as following:

1. Create a formal specification of Agent Modeling Language that provides a precise and unambiguous description of AML and can lead to additional investigation in its improvement.

2. Formally specify an abstract multi-agent system by means of the concepts of AML. This outline can be used in further research in MAS theory.

### 1.2 Structure of Thesis

The remainder of the thesis is structured as follows:

Chapter 2 *Introduction to AML* (p. 3) presents an introduction to Agent Modeling Language. Chapter 3 *Object-Z in Shortcut* (p. 4) forms a brief introduction into formal specifications and Object-Z notation.

*Part I: Solution summary* (p. 6) – This part summarizes the process of getting to the results, which are presented in Part II and Part III. Chapter 4 *Transformation of the AML Metamodel* (p. 7) describes the process of transformation of the AML Metamodel. Chapter 5 *Object-Z and the Formal Mapping* (p. 9) shows the transformation process by means of formal mapping method, and presents an abstract UML class metamodel.

*Part II: Formal Specification of AML* (p. 19) – This part contains formal specification of AML Metamodel using Object-Z specification language. Chapter 6 *Overview of used UML classes* (p. 20) familiarize the reader with an enumeration of UML classes that were used in the second part of this thesis. Chapter 7 *Organization of the AML Specification* (p. 21) captures the overall package structure of AML Metamodel. Chapters 8 *Architecture* (p. 23), 9 *Behaviors* (p. 38), 10 *Mental* (p. 69), 11 *Ontologies* (p. 83), and 12 *Model Management* (p. 85) contain Object-Z specification of all packages from the AML Kernel (Architecture, Behaviors, Mental, Ontologies and Model Management), their sub-packages and metaclasses. In Chapter 13 *UML Extension for AML* (p. 86) are presented the AML-related extensions of UML.

*Part III: Abstract Multi-Agent Framework* (p. 88) - This part presents conceptual AML metamodel originated in [1]. Chapter 14 *Concepts of AML* (p. 89) describes components that form the framework.

*Part IV: Summary of Achievements* (p. 117) - This part provides a summary of the achieved results in chapter 15 *Conclusions and Future Works* (p. 118).
Chapter 2

Introduction to AML

The Agent Modeling Language (AML) [2] is a semi-formal visual modeling language for specifying, modeling and documenting systems that incorporate concepts drawn from MAS theory. It was required to overcome the deficiencies of the current state-of-the-art and practice in the area of MAS modeling languages, namely: insufficient documentation of modeling languages, using proprietary and/or non-intuitive modeling constructs, limited scope, mutual incompatibility, insufficient support by CASE tools, etc. AML is intended to be a ready-to-use, complete and highly expressive modeling language suitable for the industrial development of real-world software solutions based on multi-agent technologies.

The starting point of development of AML was to obtain necessary know-how from the area of MAS and agent-oriented modeling in particular. Apart from study relevant theories, specification and modeling approaches, abstract MAS models, technologies, and available agent-based solutions, the main source of inspiration was drawn from existing agent-oriented modeling languages. Based on analysis of the modeled MAS aspect, R. Cervenka in [2] defined the basic MAS modeling concepts and created the MAS metamodel, which forms a conceptual basis for the design of AML.

In combination with the UML 2.0 metamodel, the previously defined MAS concepts were used to define the AML modeling constructs. The abstract syntax and semantics of AML was specified in the AML metamodel. Based on the metamodel the language’s notation was also defined and used to specify its concrete syntax. The AML metamodel and notation represent the core of the language specification.

Besides this, the author of [2] also extended the basic set of UML diagram types with additional ones, to provide agent-specific views of the system model. Another achievement of AML is the definition of a set of operators extending the OCL Standard Library [13] with operators from modal logic, deontic logic, temporal logic, dynamic logic, epistemic logic, BDI logic, etc. These operators allow the specification of OCL constraints based on different types of modal family logics that provide more natural, and commonly used, means for specification of MASs.

AML represents a consistent framework for modeling applications that embody and/or exhibit characteristics of multi-agent systems. It integrates best modeling practices and concepts from existing agent oriented modeling and specification languages into a unique framework built on the foundations of UML 2.0 and OCL 2.0. AML is also specified in accordance with the OMG modeling frameworks MOF 2.0 and Model-Driven Architecture (MDA). [1]

For more details we refer the reader to Radovan Cervenka and Ivan Trencansky [1].
Chapter 3

Object-Z in Shortcut

G. Smith in [18] writes – “Formal specification is the first step in the formal development of a software system. It is followed by a series of steps involving verification and refinement which lead to an eventual implementation. The primary role of the formal specification is to provide a precise and unambiguous description of the system as a basis for these subsequent steps.”

The uses of a formal specification are basically as following. A formal specification allows the system designer to verify important properties, resolve ambiguities and detect design errors before system development begins. Without a formal specification, a system would have to be extensively tested after implementation. This alternative is not only expensive, since on failing the tests the system may need to be reimplemented, but also can never guarantee reliable behavior.

To enable verification of system properties and refinement towards an implementation, a language for formal specification must be mathematically based. Usually this basis is expressed algebraically or in set theory and logic. A formal specification language must also have a well-defined syntax and semantics. A wide range of formal specification languages have been proposed. Most of these languages can be classified as either property-oriented (e.g. Clear [22], OBJ [23]) or model-oriented (e.g. Z [20], VDM [24]). Property-oriented languages describe a system implicitly by stating its properties whereas model oriented languages construct an explicit model of the system. Some specification languages do not belong to just one of the above classes. For example, the specification language LOTOS [25] has two distinct parts: a process algebra based on CCS and an abstract data type language based on the algebraic specification language ACT ONE [26].

The Object-Z specification language also combines two techniques. Being an extension to Z, it is primarily a state-based language but it also has a temporal logic component used to capture liveness properties.

G. Smith defines Object-Z in [18] as follow – “Object-Z is an extension of Z in which the existing syntax and semantics of Z are retained and new constructs are introduced to facilitate specification in an object-oriented style. The major extension in Object-Z is the class schema which captures the object-oriented notion of a class by encapsulating a single state schema with all the operation schemas which may affect its variables. The class schema is not simply a syntactic extension but also defines a type whose instances are objects.”

Briefly said – Object-Z (OZ) is an extension to the ISO-standardized mathematically-based specification language Z [19,20] that adds support for object-oriented constructs: classes, attributes, operations, object relationships, and inheritance.”
An Object-Z class schema, often referred to simply as a class, is represented syntactically as a named box with zero or more generic parameters. In this box there may be local type and constant definitions, at most one state and associated initial state schema and zero or more operations. A class may also include the names of inherited classes and history invariants for capturing liveness properties. The basic structure of a class is depicted in Fig. 3.1.

![Figure 3.1: Basic structure of a class in Object-Z](image)

For more details see [18, 21].
Part I

Solution Summary
Chapter 4

Transformation of the AML Metamodel

From the beginning of our work we have considered to use automatic conversion of the AML metamodel to OZ schemes. The main benefit from this approach would be correctness, reliability and effectiveness of transformation.

J.G. Süss et al. presents in [6] a practical application of MDA and reverse engineering based on a domain-specific modeling language. A well defined metamodel of a domain-specific language is useful for verification and validation of associated tools. Authors of [6] applied this approach to SIFA\(^1\), a security analysis tool. SIFA has evolved as requirements have changed and its metamodel was not defined. Hence, testing SIFA’s correctness was difficult. A formal metamodeling approach to develop a well-defined metamodel of the domain was introduced. Initially, J.G. Süss et al. developed a domain model in EMF by reverse engineering the SIFA implementation. Then they transformed EMF to Object-Z using model transformation. Finally, they completed the Object-Z model by specifying system behavior. The outcome is a welldefined metamodel that precisely describes the domain and the security properties that it analyses. It also provides a reliable basis for testing the current SIFA implementation and forward engineering its successor.

The common notion of Model-Driven Architecture [12] is one of gradual refinement of models from a platform-independent to a platform-specific model. The starting point of the process is an abstract specification of the system; the destination is an executable system. In [6] is described an experience which runs contrary to that received notion: an existing application is gradually turned into a formal specification: A process of reverse-MDA.

While the authors of [6] used a system specific reverse-MDA, it was sufficient to inspire us to generalize this process for our purpose. Fig. 4.1 shows a generalized reverse-MDA. Existing System is converted using reverse engineering into UML Model and using XMI parsing into Object Model. Once the structure and behavior are known, we can follow to construct formal Object-Z Model and its printable \LaTeX{} representation. In our case, the system stands for the AML metamodel which needs to be transferred into a printable version of its OZ representation. Therefore, we needed to find an acceptable way of doing so.

\(^1\)SIFA stands for Security Information Flow Analyser and is a part of an information security project developed by Tim McComb and Luke Wildman.
J.G. Süß et al. in their paper needed to build an initial version of their metamodel quickly. Their approach could be described as a toolchain involving Rational Rose and its XML-DTD importer, Eclipse EMF [15] and its Rational Rose importer, the Tefkat QVT transformation engine [16], a text-storage module, and the Community Z Tools (CZT) suite [17]. They reverse-engineered the data structures of their system called SIFA into a UML class diagram, using a DTD-to-UML converter, which generated a model based on an XML DTD Profile and removed the profile to turn the DTD model into a general UML model. This model was visualized in several diagrams and reworked and refined with the aid of SIFA’s author to yield a first draft metamodel. This metamodel was imported into the Eclipse Metamodeling Framework (EMF). They used then model transformation to convert the EMF representation of SIFA into an instance of a metamodel of the Object-Z language. Both Ecore and Object-Z are object-oriented modeling languages and share the common concepts of object-orientation: classes, attributes, operations, object relationships and inheritance. Thus, transformation between the languages was straightforward. The transformation system that was used is DSTC’s Tefkat. Tefkat uses a declarative language which is expressive and backed by a prolog-based solver. Hence its formalism is well-suited to directly encode the formal correspondences between UML static structure models and Object-Z static structure models, as laid out in [7]. The specification was completed by enriching it with a behavioral description. There was also a need to visualize instances of the model. Object-Z is a superset of the Z notation, which has a standard LATEX concrete syntax. Authors of [6] therefore created a converter from the XMI representation of an Object-Z instance to its LATEX representation. With a complete and formal domain metamodel whose instances could be converted to LATEX they were able to tap into the resources of the Object-Z community: the Community Z Tools project (CZT) [17]. Among the CZT tools are editors, textual layout tools for HTML and print-media, a type-checker, and connectors for external model-checking tools. They used CZT to type-check the Object-Z and add more refined constraints and behaviors.

Even thou we were inspired by the aforementioned transformation approach described by J.G. Süß et al., our approach was slightly different. The EMF representation of the AML Metamodel was constructed in Eclipse [15]. Next step would be to use DSTC’s Tefkat transformation engine to obtain Object-Z specification. Since the transformation rules were available only partially, we made contact with Dr. Soon-Kyeong Kim from the University of Queensland in Australia, who helped us with our transformation for a while. But due to organization problems this part of work was not satisfactory finished. There was no other alternative for us – using [5], [6], and [8] we were able to make the transformation manually. As it has been stated above, the transformation between Ecore and Object-Z modeling languages was straightforward, but not exhaustive. We required quite a lot of time to make it complete. Chapter 5 Object-Z and the Formal Mapping (p. 9) explains this process in more detail.
Chapter 5

Object-Z and the Formal Mapping

This chapter introduces the formal mapping between UML models and Object-Z specifications and explains, how this mapping works.

5.1 Formal Mapping Between UML Models and Object-Z Specifications

This approach has been presented by Soon-Kyeong Kim and David Carrington in [5]. The goal of their work was to provide a formal basis for the syntactic structures and semantics of UML modeling constructs and to provide a sound mechanism for reasoning about UML models. To achieve this goal, they first gave a formal description for UML modeling constructs using Object-Z classes. Second, they translated UML modeling constructs to Object-Z constructs for a rigorous analysis of UML models. This was achieved by a definition of an abstract metamodel for UML and Object-Z. In the metamodel, the abstract syntax and semantics of core modeling constructs are grouped together into Object-Z classes. For better understanding the UML class diagrams were used to show the structure of both UML and Object-Z modeling constructs. Given the formal description for UML constructs and Object-Z constructs, the UML constructs are translated to Object-Z constructs. The translation process is described formally in terms of mapping functions. The scope of [5] is restricted only to the UML class constructs and class diagrams.

Following schemes represents UML class metamodel and are not equivalent with UML Infrastructure 2.0 defined in [10], but we have decided to add it. We believe that the reader will gain this way a better insight in the Object-Z. All the conditions that refer to [10,11] (e.g. see section 8.1.2) in the second part of our work have been declared as if the UML 2.0 metamodel in OZ did exist. To our knowledge this didn’t happen yet.

[Name]

Name is a given set from which the names of all classes, attributes, operations, operation parameters, associations and roles are drawn.
5.1. Formal Mapping Between UML Models and Object-Z Specifications

An Object-Z class Type is a meta type, from which all possible types in UML such as object types, basic types (integer and string) and so on can be derived. Each type has a name and contains a collection of its own features: attributes and operations. Thus, a circled c, which models a containment relationship in Object-Z is attached to the types of attributes and operations.

VisibilityKind ::= private | public | protected

Visibility in UML can be private, public, or protected.

Attributes and parameters are also defined as follows. Variable multiplicity in Attribute describes the possible number of data values for the attribute that may be held by an instance.

Within an operation, parameter names should be unique.

∀ a1, a2 : attributes • a1.name = a2.name ⇒ a1 = a2
∀ op1, op2 : operations •
  (op1.name = op2.name ∧ #op1.name = #op2.name ∧
   ∀ i : 1..#op1.parameters •
     op1.parameters(i).name = op2.parameters(i).name ∧
     op1.parameters(i).type = op2.parameters(i).type) ⇒ op1 = op2
5.1. Formal Mapping Between UML Models and Object-Z Specifications

With these classes, we define an Object-Z class `Class` as follows. Since a class is a type, it inherits from `Type`. Attribute names defined in a class should be different and operations should have different signatures. The class invariant formalizes these properties.

\[
\text{Boolean} ::= \text{true} \mid \text{false}
\]

Boolean represents boolean data type.

\[
\text{AggregationKind} ::= \text{none} \mid \text{aggregate} \mid \text{composite}
\]

The Object-Z class `AssociationEnd` is a formal description of association ends. It has a role name, a multiplicity constraint, an attached class and attributes for describing aggregation and navigability. The multiplicity constraint describes a range of nonnegative integers denoting the allowable cardinality constraints for instances of the class attached to the other end. The variable aggregation can take the values none, aggregate, or composite. The variable navigability can be true or false. The constraints in the predicate state that a multiplicity cannot be 0 and for composition, the multiplicity of the composite end can be no more than one.

\[
\text{Association}
\]

A binary association has a name and exactly two association ends. An Object-Z class `Association` is a formal description of binary associations.

The constraints in the predicate state the core properties of association:

- Each role name must be different.
• For aggregation and composition, there should be an aggregate or a composite end and the other end is therefore a part and should have the aggregation value of none. We assume that e1 is the composite or aggregate.

• For an association or an association class, the role name at an association end should be different from the attribute names of the class attached to the other end.

• An association name should be unique in the combination of its attached classes.

\[
\text{AssocClass} \\
\text{Class} \\
\text{Association}
\]

\[
e1.\text{aggregation} = \text{none} \land e2.\text{aggregation} = \text{none} \\
\text{self} \notin \{e1.\text{attachedClass}, e2.\text{attachedClass}\} \\
\{a : \text{attributes} \bullet a.\text{name}\} \cap \{e1.\text{rolename}, e2.\text{rolename}\} = \emptyset
\]

An association class inherits from a class and an association. We define an Object-Z class AssocClass inheriting from Class and Association.

The constraints describe well-formedness rules for association classes:

• the aggregation value of both association ends is none,
• an association class cannot be defined between itself and something else, and
• the role names and the attribute names do not overlap.

\[
\text{Generalization}
\]

\[
\text{super : } \downarrow\text{Class} \\
\text{sub : } \downarrow\text{Class} \\
\{g : \text{Generalization} \bullet (g.\text{super}, g.\text{sub})\}^* \cap \text{id}(\downarrow\text{Class}) = \emptyset
\]

In UML, a generalization describes a taxonomic relationship between objects, in which objects of the superclass have general information and objects of the subclasses have more specific information [10, 11]. This relationship is defined with an Object-Z class named Generalization. In the class, two variables, super and sub are declared to represent the superclass and the subclass involved in a generalization. The constraint prohibits any circular inheritance.
A UML class diagram is a collection of classes including association classes, associations and generalizations between these classes. Classes should have unique names within the class diagram. The following Object-Z class is a formal description of UML class diagrams. The constraints describe that:

- Classes that are involved in associations or association classes should be classes in the diagram.
- Classes involved in generalizations should be classes in the diagram.

The reader looking for more details is referred to [5].

5.2 Usage of the Formal Mapping

In this section we show the formal mapping described earlier.

Following review was adopted from David Roe, [8].

The translation of the UML class diagrams (without OCL) into Object-Z structures is presented here using examples based on the UML diagram given in Fig. 5.1.

![Figure 5.1: An UML class diagram for persons and bank accounts](image)

5.2.1 Mapping Classes

Consider the simple UML class diagram of Fig. 5.1. Ordinary UML classes like `Account` and `Person` may be mapped into an Object-Z class construct of the same name, with class features transcribed to the enclosed schemas defining state variables, constants and class operations. Features marked public (+) are included within the class construct visibility list, while those that are unadorned or marked private (−) are not.
5.2. Usage of the Formal Mapping

5.2.2 Mapping Attributes

UML attributes are mapped as variables of the same name, declared within the state schema of the corresponding Object-Z class construct or within the separate constant definition schema when marked with UML’s \{frozen\} property string. Attribute type declarations are required for translation to Object-Z, which supports a range of well known domains corresponding to most basic programming types.

User-defined classes may also be employed as types within UML models and Object-Z specifications; for example, a person’s sex might have been enumerated (male, female) within the UML model corresponding to the definition of a named domain, \(\text{Sex} = \{\text{male}, \text{female}\}\) in Object-Z.

Attributes with multiplicities greater than one may be mapped as finite sequences of the base UML type, combined with a cardinality restriction. A person’s dateOfBirth attribute therefore corresponds to the declaration of the state variable \(\text{dateOfBirth} : \text{seq Z}\) and predicate \(\#\text{dateOfBirth} = 3\). Derived attributes, marked (/) in the UML, are distinguished from primary variables within Object-Z schema through the \(\Delta\) separator.

5.2.3 Mapping Operations

UML class operations may be translated as individual Object-Z operation schema with the same name, with parameters and return values mapped as input and output communication variables adorned (?) and (!) respectively. Although parameter names are optional within the UML, and return values are not named, both must be supplied for the purposes of translation to Object-Z. As with attributes, UML operations marked public are included within the class construct visibility list. Based on the discussion so far, Fig. 5.2 provides a translated class skeleton for class \(\text{Account}\).

5.2.4 Mapping Associations

Associations may be represented through the instantiation of additional state attributes in Object-Z, depending upon the navigability specified across the UML association line. Fig. 5.1 depicts navigability from class \(\text{Person}\) to class \(\text{Account}\), implying an additional attribute within the Object-Z class \(\text{Person}\). Its name is mapped from the target class rolename (since none is specified in this example, \(\text{account}\) by default) and its type is the power set of the target class. Bi-directional associations are mapped as if they were two separate uni-directional associations. Association multiplicities are reflected in additional state axioms constraining the size of such sets, in this case \(0 \leq \text{account} \leq 3\). Fig. 5.3 provides a mapping for class \(\text{Person}\), reflecting the navigable association with class \(\text{Account}\). The class association management operations are described later.

5.2.5 Mapping Aggregation and Composition

Translation of aggregations therefore proceeds much as for ordinary associations, with the compound class construct containing an additional state variable of type power set of the part class. UML composition, by contrast, implies that instances of the part class may belong to just one instance of the compound class. Mapping is straightforward in that Object-Z provides a notational shorthand (©) denoting unshared containment. Composition between an account and the transactions made on that account, for example, may
5.2. Usage of the Formal Mapping

be captured through the declaration of a state variable \( \text{transactions} : \text{Transaction} \) in the state schema of class Account.

5.2.6 Mapping Association Classes

Association classes permit class like features to be added to UML associations. Such classes may be formalized in Object-Z as described above, but with the addition of two state variables corresponding to the rolenames and types of the classes participating in the association. Depending upon the navigability specified across the association line, the participating class constructs will contain an additional attribute whose type is a power set of the association class, and constrained in size by the multiplicity specified at the opposite association end.
5.3. Additional Functions

5.2.7 Mapping Generalization

Mapping of UML generalization is straightforward in that Object-Z provides a simple notation denoting inheritance, with child classes naming inherited classes just below their visibility list. Specialized subclass features may then be mapped as described earlier.

Fact about inheritance in OZ [21, p. 13]: The visibility list (denoted by $\uparrow$) of a class is not inherited, and must be respecified. Visible features may be removed from the interface, invisible may be made visible. In our work can often be seen $\uparrow (\ldots, \text{list_of_visible_items})$. Using “…” we state that also the visible items from the parent classes are inherited.

5.3 Additional Functions

In this section we present some additional functions that are used mainly in the second part of this thesis.

To define constraints, authors of [1] used in their work the UML 2.0 OCL Specification [13]. In our work we use some similar functions, but first let us present the most important OCL functions:

- $\text{oclIsKindOf}(t : \text{OclType}) : \text{Boolean}$ – The $\text{oclIsKindOf}$ property determines whether $t$ is either the direct type or one of the supertypes of an object.

- $\text{conformsTo}(c : \text{Classifier}) : \text{Boolean}$ – The $\text{conformsTo}$ operation is defined on $\text{Classifier}$. It evaluates to true, if the self $\text{Classifier}$ conforms to the argument $c$.

- $\text{oclAsType}(\text{OclType})$ – This operation results in the same object, but the known type is the argument $\text{OclType}$. When it is certain that the actual type of the object is the subtype, the object can be re-typed using this operation.

- $\text{includesAll}(c2 : \text{Collection}(T)) : \text{Boolean}$ – This operation answers following question: “Does self contain all the elements of $c2$ ?”

We use similar functions in our OZ specification of AML:

- $\text{isKindOf} : \downarrow\text{OZType} \times \downarrow\text{OZType} \rightarrow \text{Boolean}$ – Function determines whether the value in first argument is either the direct type or one of the supertypes of the second argument.

- $\text{conformsTo} : \downarrow\text{Classifier} \times \downarrow\text{Classifier} \rightarrow \text{Boolean}$ – Function determines whether the value in first argument conforms to value in second argument.

- $\text{asType} : \downarrow\text{OZType} \times \downarrow\text{OZType} \rightarrow \downarrow\text{OZType}$ – This function re-types type given in the first argument to type given in second argument and returns modified type. See $\text{oclAsType}$ operation.

- $\text{includesAll} : \downarrow\text{Collection} \times \downarrow\text{Collection} \rightarrow \text{Boolean}$ – Function determines whether the collection given in first attribute contains all elements from the second collection.
5.4. Example of Mapping

For instance, roleAttribute = self.ownedAttribute → select(oclIsKindOf(RoleProperty)) can be expressed in OZ as follows:

\[ \forall ra : roleAttribute \bullet \forall oa : self.ownedAttribute \mid \text{isKindOf}(oa, \text{RoleProperty}) = \text{true} \bullet \]

\[ ra = oa \]

The roleAttribute is equal to all ownedAttributes that are of the kind RoleProperty. In OZ we express this fact similar. We say, that all roleAttribute objects are equal to all ownedAttribute objects, which are of the kind RoleProperty. As we can see, the isKindOf function is used in similar way as the oclIsKindOf function. The only difference is the first argument.

The ↓OZType comes from the Object-Z metamodel defined by Soon-Kyeong Kim and David Carrington in [5]. It is an abstract class from which all possible types in Object-Z can be derived. Meaning of ↓ notation can be explained as follows – “In Object-Z, a variable can be declared, explicitly, to be an object of any class in a particular inheritance hierarchy. For example, if C is a class then the declaration c : ↓C declares the object c to be of class C or any class derived from C by inheritance.” [18]

The Classifier and Collection types are defined in [10,11].

As was stated before, to present the complete OZ specification of AML, we would require the UML metamodel in OZ, but this specification is provided only partially and mainly is not defined as in [10,11]. In this chapter (section 5.1, p. 9) we already presented such metamodel. Also, to define the afore mentioned functions, we would need to refer to the Object-Z metamodel. There exists a specification, which can be found in [5]. But to make things work properly, we would need greatly to extend our work. Naturally, this would lead us beyond the scope of this thesis.

5.4 Example of Mapping

Figure 5.4 shows a MentalProperty class (section 10.1.8, p. 73) in the AML Metamodel [1, p. 268].

Using the formal mapping we can transform this UML class diagram into following OZ schema (Fig. 5.5). The ∀ o : association • o.mentalMemberEnd = self condition in MentalProperty and the ∀ o : mentalMemberEnd • o.association ∈ self condition in MentalAssociation (section 10.1.9, p. 74) ensure the consistency of the bi-directional relationship.

For a complete definition of MentalProperty class containing also mapped OCL constraints, we refer the reader to section 10.1.8 on page 73.
5.4. Example of Mapping

Figure 5.4: **MentalProperty** class in AML Metamodel

**MentalProperty**

\[(\ldots, \text{degree}, \text{association}, \text{type}, \text{mentalConstraint})\]

\[\text{Property}\]

\[
\begin{align*}
\text{degree} & : \text{seq} \quad \text{ValueSpecification} \\
\text{association} & : \mathbb{P} \quad \text{MentalAssociation} \\
\text{type} & : \mathbb{P} \quad \text{MentalClass} \\
\text{mentalConstraint} & : \mathbb{P} \quad \text{MentalConstraint} \\
\end{align*}
\]

\[
\begin{align*}
\#\text{degree} \leq 1 \\
\#\text{association} \leq 1 \\
\forall o : \text{association} \cdot o.\text{mentalMemberEnd} = \text{self} \\
\#\text{type} \leq 1
\end{align*}
\]

Figure 5.5: **MentalProperty** class schema without mapped OCL constraints
Part II

Formal Specification of AML
Chapter 6

Overview of used UML classes

The following classes of the UML 2.0 metamodel were used in our formal specification of AML. Boldly are marked partially defined metaclasses that can be found in section 5.1 on page 9, but they mainly serve for demonstrative purposes and are, as we have notified (section 5.1, p. 9), not equivalent with UML 2.0 metamodel [10]. The reader is at this place referenced to [10,11] for more details. We also remind him that in this work all UML metaclasses were used as if they would be specified and would correspond to [10,11].

Chapter 7

Organization of the AML Specification

In order to improve the readability and comprehension of the specification, the AML Metamodel 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.

The second part of this thesis is organized as follow:

• Chapters 8 Architecture (p. 23), 9 Behaviors (p. 38), 10 Mental (p. 69), Chapters 11 Ontologies (p. 83), and 12 Model Management (p. 85) hold these conventions:
  o Each chapter represents a package from AML Metamodel package (see Fig. 7.1).
  o Each section stands for a package on the lowest level of package hierarchy.

• Chapter 13 UML Extension for AML (p. 86) fulfil similar precondition:
  o Each chapter stands for a package on the lowest level of package hierarchy.

• Each section is described in following matter:
  o A short informal definition of the metaclass.
  o A brief explanation of the reasons why a given metaclass is defined within AML.
  o A formal specification of presented AML metaclass depicted in a schema-like form.
  o A natural language explanation of presented Object-Z class schema.

The AML Metamodel is logically structured according to the various aspects of MAS abstractions. All packages and their content are described in the following chapters. The overall package structure of the AML metamodel is depicted in Fig. 7.1.

The structure of this chapter has been mainly inspired by [1], which serves as a rational reference between our work and [1, Part III: AML Specification].
Figure 7.1: Overall package structure of the AML metamodel
Chapter 8

Architecture

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

8.1 Entities

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.

8.1.1 EntityType

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 AgentExecution-Environments (section 8.6.1, p. 34), and can be mobile (section 9.6.3, p. 66). For more details see [1, p. 138].

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

\[
\text{EntityType} \subseteq \text{Type}
\]

\[
\text{EntityType} = \emptyset
\]

EntityType is an abstract Object-Z class, which inherits from Type. Abstractness is expressed in following condition in the state schema: EntityType = ∅. In Object-Z class types are interpreted as disjoint sets of object identities, where such identities represent possible unique instantiations. By default there are an infinite (although countable) number of possible instantiations of classes because these sets are unbounded, but above EntityType is constrained to be empty (∅ is the empty set). This ensures that EntityType cannot be instantiated.
8.1.2 BehavioralEntityType

*BehavioralEntityType* is an abstract specialized *EntityType* used to represent types of entities which have the features of *BehavioredSemiEntityType* and *SocializedSemiEntityType*, and can play entity roles (see sections 8.5.6 and 8.5.7). Instances of *BehavioralEntityTypes* are referred to as behavioral entities. For more details see [1, p. 138].

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

\[
\begin{align*}
\text{BehavioralEntityType} & \rightarrow (\ldots, \text{roleAttribute}) \\
\text{BehavioredSemiEntityType} & \\
\text{SocializedSemiEntityType} & \\
\text{EntityType} &
\end{align*}
\]

\[\Delta \]

\[\begin{align*}
\text{roleAttribute} : \forall \text{RoleProperty} \\
\text{BehavioralEntityType} = \emptyset \\
[1] \forall ra : \text{roleAttribute} \bullet \\
\quad \forall oa : \text{self.ownedAttribute} | \text{isKindOf}(oa, \text{RoleProperty}) = \text{true} \bullet \\
\quad ra = oa
\end{align*}\]

*BehavioralEntityType* is an abstract Object-Z class that inherits from *EntityType, BehavioredSemiEntityType*, and *SocializedSemiEntityType*. It comprises of roleAttribute, which is derived attribute. In Object-Z all attributes below \(\Delta\) are derived attributes. Invariant [1] formalizes the fact, that all *roleAttribute* instances are equal to all *ownedAttributes* instances that are of *RoleProperty* kind.

8.1.3 AutonomousEntityType

*AutonomousEntityType* is an abstract specialized *BehavioralEntityType* and *MentalSemiEntityType* 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*. Instances of *AutonomousEntityTypes* are referred to as autonomous entities. For more details see [1, p. 139].

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

\[
\begin{align*}
\text{AutonomousEntityType} & \\
\text{BehavioralEntityType} & \\
\text{MentalSemiEntityType} &
\end{align*}
\]

\[
\text{AutonomousEntityType} = \emptyset
\]
8.2 Agents

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

8.2.1 AgentType

AgentType is a specialized AutonomousEntityType used to model 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 object-oriented paradigm defines as an entity having identity, status and behavior; not narrowed to an object-oriented programming concept) having at least the following additional features:

- 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, for instance, associations, generalizations, or dependencies, with their standard semantics, as well as inherited AML-specific relationships described in further sections. For more details see [1, p. 140].

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

\[
\text{AgentType} \\
\text{AutonomousEntityType}
\]

Object-Z class AgentType inherits from AutonomousEntityType.

8.3 Resources

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

8.3.1 ResourceType

ResourceType is a specialized BehavioralEntityType used to model types of resources contained within the system. 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). For more details see [1, p. 142].

ResourceType is introduced to model types of resources in multiagent systems.

\[
\text{ResourceType} \\
\text{BehavioralEntityType}
\]
8.4 Environments

The Environments package defines the metaclasses used to model system internal environments (for definition see section 8.4.1) of multi-agent systems.

8.4.1 EnvironmentType

EnvironmentType is a specialized AutonomousEntityType used to model types of environments, i.e. the logical and physical surroundings of entities which provide conditions under which those 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, and also those principles and processes (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. For more details see [1, p. 143].

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

\[
\begin{align*}
\text{EnvironmentType} & \quad \text{AutonomousEntityType} \\
\end{align*}
\]

EnvironmentType is a specialized AutonomousEntityType class.

8.5 Social Aspects

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.
8.5. Social Aspects

8.5.1 OrganizationUnitType

OrganizationUnitType is a specialized EnvironmentType 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 environments that specify the social arrangements of entities in terms of structures, interactions, roles, constraints, norms, etc. For more details see [1, p. 147].

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

\[ \text{OrganizationUnitType} \subseteq \text{EnvironmentType} \]

Object-Z class OrganizationUnitType inherits from EnvironmentType class.

8.5.2 SocializedSemiEntityType

SocializedSemiEntityType is an abstract specialized Class (from UML), a superclass to all metaclasses which can participate in SocialAssociations and can own SocialProperties. There are two direct subclasses of the SocializedSemiEntityType: BehavioralEntityType and EntityRoleType. SocializedSemiEntityTypes represent modeling elements, which would most likely participate in CommunicativeInteractions. 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. For more details see [1, p. 149].

SocializedSemiEntityType is introduced to define the features common to all its subclasses.
8.5. Social Aspects

\[ \text{SocializedSemiEntityType} = \langle \ldots, \text{supportedAcl}, \text{supportedCl}, \text{supportedEncoding}, \text{supportedOntology}, \text{socialAttribute} \rangle \]

\[ \text{supportedAcl} : \text{seq} \ \text{ValueSpecification} \]
\[ \text{supportedCl} : \text{seq} \ \text{ValueSpecification} \]
\[ \text{supportedEncoding} : \text{seq} \ \text{ValueSpecification} \]
\[ \text{supportedOntology} : \text{seq} \ \text{ValueSpecification} \]
\[ \Delta \]
\[ \text{socialAttribute} : \mathbb{P} \ \text{SocialProperty} \]

\[ \forall \text{sa} : \text{socialAttribute} \bullet \]
\[ \forall \text{oa} : \text{self.ownedAttribute} \mid \text{isKindOf} (\text{oa}, \text{SocialProperty}) = \text{true} \bullet \]
\[ \text{sa} = \text{oa} \]

\text{SocializedSemiEntityType} is an abstract Object-Z class, that inherits from \text{Class}. All \text{SocializedSemiEntityType}’s attributes are visible (they all belong in the visibility list). Invariant [1] formalizes the fact, that all \text{socialAttribute} instances are equal to all \text{ownedAttributes} instances that are of \text{SocialProperty} kind.

8.5.3 SocialProperty

\text{SocialProperty} is a specialized \text{ServicedProperty} used to specify social relationships that can or must occur between instances of its type and:

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

\text{SocialProperty} can be only of a \text{SocializedSemiEntityType} type. \text{SocialProperties} can be owned only by:

- \text{SocializedSemiEntityTypes} as attributes, or
- \text{SocialAssociations} as member ends.

When a \text{SocialProperty} is owned by a \text{SocializedSemiEntityType}, it represents a social attribute. In this case the \text{SocialProperty} can explicitly declare a social role of its type in regard to the owning class. For more details see [1, p. 151].

\text{SocialProperty} is introduced to model social relationships between entities in multi-agent systems.
SocialProperty class inherits from ServicedProperty. The size of socialRole and association set is at most one. The attribute association in the SocialProperty class coresponds to an attribute association in the SocialAssociation class, indicating a bi-directional relationship between SocialProperty and SocialAssociation. The consistency of the bi-directional relationship is ensured via the predicate ∀ o : association • self ∈ o.memberEnd in SocialProperty and the predicate ∀ o : memberEnd • o.association ∈ self in SocialAssociation. Similar conditions can be found in some undermentioned Object-Z classes. Condition [1] says that when association set not empty and when socialRole is peer, then the socialRoles of all other member ends must be set to peer as well. Invariant [2] express similar condition, but states also that the selected memberEnd is not equal to SocialProperty self. Condition [3] says that 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.

8.5.4 SocialRoleKind

SocialRoleKind is an enumeration which specifies allowed values for the socialRole meta-attribute of the SocialProperty. 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). For more details see [1, p. 154].

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

\[
\text{SocialRoleKind ::= peer \mid superordinate \mid subordinate}
\]
8.5. Social Aspects

In Object-Z, SocialRoleKind is defined as enumeration, which has peer, superordinate, and subordinate values.

8.5.5 SocialAssociation

SocialAssociation is a specialized Association (from UML) used to model social relationships that can occur between SocializedSemiEntityTypes. It redefines the type of the memberEnd property of Association to SocialProperty. An instance of the SocialAssociation is called social link.

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

\[
\text{SocialAssociation} \\
[\ldots, \text{memberEnd}] \\
\text{Association}
\]

\[
\text{memberEnd} : \mathbb{P}\text{SocialProperty} \\
\#\text{memberEnd} \geq 2 \\
\forall o : \text{memberEnd} \bullet o.\text{association} \in \text{self}
\]

SocialAssociation class inherits from Association class. The size of memberEnd set is greater than two. SocialAssociation is in bi-directional relationship with SocialProperty.

8.5.6 EntityRoleType

EntityRoleType is a specialized BehavioredSemiEntityType, MentalSemiEntityType, and SocializedSemiEntityType, used to represent a coherent set of features, behaviors, participation in interactions, and services offered or required by BehavioralEntityTypes 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 or Context. Each EntityRoleType should be realized by a specific implementation possessed by a BehavioralEntityType which may play that EntityRoleType. 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. 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 8.1.2 for details). The entity role exists only while a behavioral entity plays it. For more details see [1, p. 156].

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

\[
\text{EntityRoleType} \\
\text{SocializedSemiEntityType} \\
\text{BehavioredSemiEntityType} \\
\text{MentalSemiEntityType}
\]
8.5. Social Aspects

EntityRoleType is an Object-Z class, which inherits from SocializedSemiEntityType, BehavioralSemiEntityType, and MentalSemiEntityType.

8.5.7 RoleProperty

RoleProperty is a specialized Property (from UML) used to specify that an instance of its owner, a BehavioralEntityType, can play one or several entity roles of the specified EntityRoleType. The owner of a RoleProperty is responsible for implementation of all Capabilities, StructuralFeatures and metaproperties defined by SocializedSemiEntityType 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 a RoleProperty constrains the number of entity roles of a given type that the particular behavioral entity can play concurrently. For more details see [1, p. 158].

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

\[
\text{RoleProperty} \\
| (\ldots, \text{association}) \\
\text{Property} \\
\text{association} : \mathbb{P} \text{PlayAssociation} \\
\#\text{association} \leq 1 \\
[1] \text{self.aggregation} = \text{composite}
\]

RoleProperty class inherits from Property class. The association set is greater than one. Invariant [1] formalizes the fact that aggregation attribute of the RoleProperty class is composite.

8.5.8 PlayAssociation

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

- If there are more than one PlayAssociations attached to an EntityRoleType then an implicit constraint applies, stating that no more than one PlayAssociation link can exist at any given moment. These constraints are implicit and thus not shown in diagrams.
8.5. Social Aspects

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. For more details see [1, p. 160].

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

\[
\text{PlayAssociation} \\
\bigl(\ldots, \text{roleMemberEnd}, \text{memberEnd}\bigr) \\
\text{Association}
\]

\[
\begin{align*}
\text{roleMemberEnd} & : \mathcal{P} \text{ RoleProperty} \\
\text{memberEnd} & : \mathcal{P} \text{ Property} \\
\#\text{roleMemberEnd} & = 1 \\
\forall o : \text{roleMemberEnd} \cdot o.\text{association} & \in \text{self} \\
\#\text{memberEnd} & = 2
\end{align*}
\]

PlayAssociation class inherits from Association class. The size of roleMemberEnd set is equal one. PlayAssociation is in bi-directional relationship with Property.

8.5.9 CreateRoleAction

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 into an entity role of that type, and adding this instance as a value to the RoleProperty 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).

For more details see [1, p. 162].

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

CreateRoleAction

\[ \text{(..., role, roleType, player, roleProperty)} \]

CreateObjectAction

AddStructuralFeatureValueAction

\[ \text{role : OutputPin } \circledast \]
\[ \text{roleType : EntityRoleType} \]
\[ \text{player : InputPin } \circledast \]
\[ \text{roleProperty : RoleProperty} \]

[1] \( \forall t : \text{player.type} \mid t \neq \emptyset \bullet \)
\( \text{isKindOf}(t, \text{BehavioralEntityType}) = \text{true} \)

[2] \( \forall t : \text{role.type} \mid t \neq \emptyset \bullet \)
\( \text{conformsTo}(t, \text{roleType}) = \text{true} \)

[3] \( \forall t : \text{roleProperty.type} \mid t \neq \emptyset \bullet \)
\( \forall rt : \text{roleType} \bullet \)
\( \text{conformsTo}(rt, t) = \text{true} \)

CreateRoleAction class inherits from CreateObjectAction and AddStructuralFeatureValueAction classes. The declaration of role (player) signifies that the role (player) attribute is a set of OutputPin (InputPin) instances, where that set is contained. The \( \circledast \) symbol stands for object containment in Object-Z. Following invariants must be satisfied:

[1] If the player.type of the InputPin is specified, it must be a BehavioralEntityType.

[2] If the role.type of the OutputPin is specified, it must conform to the EntityRoleType referred to by the roleType.

[3] If the roleProperty.type of the RoleProperty is specified, then the EntityRoleType referred to by the roleType must conform to it.

8.5.10 DisposeRoleAction

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. For more details see [1, p. 165].

DisposeRoleAction is introduced to model the action of disposing of entity roles by behavioral entities.
**8.6. MAS Deployment**

Disposal of a role, followed by destruction of the object it is related to.

---

\[ \text{DisposeRoleAction} \]
\[ \langle \ldots, \text{role} \rangle \]
\[ \text{DestroyObjectAction} \]

- \( \text{role} : \mathbb{P} \text{InputPin} \copyright \)
- \( \#\text{role} \geq 1 \)
- \( [1] \forall r : \text{role} \mid r\text{.type} \neq \emptyset \bullet \) \[ \text{isKindOf}(r\text{.type}, \text{EntityRoleType}) = \text{true} \]

_DisposeRoleAction_ class inherits from _DestroyObjectAction_ class. Invariant [1] express the fact, that if the _types_ of the _InputPins_ referred to by the _role_ are specified, they must be _EntityRoleTypes_.

**8.6 MAS Deployment**

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

**8.6.1 AgentExecutionEnvironment**

_AgentExecutionEnvironment_ is a specialized _ExecutionEnvironment_ (from UML) and _BehavioredSemiEntityType_, 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, for example, agent platform, or agent container. _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_ and _ServiceUsages_ respectively. Owned _HostingProperties_ 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. For more details see [1, p. 166].

_AgentExecutionEnvironment_ is introduced to model execution environments of multi-agent systems, i.e. the environments in which the entities exist and operate.
8.6. MAS Deployment

AgentExecutionEnvironment

\[(\ldots, \text{hostingAttribute})\]

ExecutionEnvironment

BehavioredSemiEntityType

\[\Delta \]

hostingAttribute : \(\mathbb{P}\) HostingProperty

[1] The internal structure of an AgentExecutionEnvironment can also consist of other attributes than parts of the type Node.

[2] \(\forall ha : \text{hostingAttribute} \bullet \forall oa : \text{self.ownedAttribute} \mid \text{isKindOf}(oa, \text{HostingProperty}) = \text{true} \bullet ha = oa\)

AgentExecutionEnvironment is an Object-Z class that inherits from ExecutionEnvironment and BehavioredSemiEntityType classes. Invariant [1] is expressed only in natural language due to absented UML metamodel. Invariant [2] express following fact – the hostingAttribute refers to all ownedAttributes of the kind HostingProperty.

8.6.2 HostingProperty

HostingProperty is a specialized ServicedProperty used to specify what EntityTypes can be hosted by what AgentExecutionEnvironments. Type of a HostingProperty can be only an EntityType. HostingProperties can be owned only by:

- AgentExecutionEnvironments as attributes, or
- HostingAssociations as member ends.

The owned meta-attribute hostingKind specifies the relation of the referred EntityType to the owning AgentExecutionEnvironment (for details see section 8.6.1). For more details see [1, p. 169].

HostingProperty is introduced to model the hosting of EntityTypes by AgentExecutionEnvironments.
HostingProperty class inherits from ServicedProperty class. Following invariants must be satisfied:

1. Every move instance refers to all clientDependencies of the kind Move.
2. Every moveFrom instance refers to all supplierDependencies of the kind Move.
3. Every clone instance refers to all clientDependencies of the kind Clone.
4. Every cloneFrom instance refers to all supplierDependencies of the kind Clone.

8.6.3 HostingKind

HostingKind is an enumeration which specifies possible hosting relationships of EntityTypes to AgentExecutionEnvironments. 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. For more details see [1, p. 172].

HostingKind is introduced to define possible values of the hostingKind meta-attribute of the HostingProperty metaclass.

\[ \text{HostingKind} ::= \text{resident} | \text{visitor} \]

HostingKind is an enumeration, which has resident and visitor values.

### 8.6.4 HostingAssociation

HostingAssociation is a specialized Association (from UML) used to specify HostingProperty in the form of an association end. It specifies that entities classified according to a hostingMemberEnd’s type (which is an EntityType) can be hosted by instances of an AgentExecutionEnvironment representing the other end type. HostingAssociation is a binary association. An instance of the HostingAssociation is called hosting link. For more details see [1, p. 172].

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

\[
\text{HostingAssociation} \longleftarrow (\ldots, \text{memberEnd}, \text{hostingMemberEnd}) \\text{Association} \\
\text{memberEnd} : \mathbb{P} \text{Property} \\
\text{hostingMemberEnd} : \mathbb{P} \text{HostingProperty} \\
\#\text{memberEnd} = 2 \\
\#\text{hostingMemberEnd} = 1 \\
\forall o : \text{hostingMemberEnd} \bullet o.\text{association} \in \text{self}
\]

HostingAssociation class inherits from Association class. HostingAssociation is in bi-directional relationship with HostingProperty.
Chapter 9

Behaviors

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

9.1 Basic Behaviors

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

9.1.1 BehavioredSemiEntityType

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

- own Capabilities,
- observe and/or effect their environment by means of Perceptors and Effectors, and
- provide and/or use services by means of ServicedPorts.

Furthermore, behavior of BehavioredSemiEntityTypes (and related features) can be explicitly (and potentially recursively) decomposed into BehavioralFragments. In addition to the services provided and used directly by the BehavioredSemiEntityType (see the service-Usage and the service Provision metaassociations inherited from the ServicedElement), it is also responsible for implementation of the services specified by all ServiceProvisions and ServiceUsages owned by the ServicedProperties and ServicedPorts having the BehavioredSemiEntityType as their type. Instances of BehavioredSemiEntityTypes are referred to as behaviored semi-entities. For more details see [1, p. 176].

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.
BehavioredSemiEntityType

\[ (... \text{behaviorFragment}, \text{ownedServicedPort}, \text{ownedPerceptor}, \text{ownedEffector}, \text{capability}) \]

ServicedElement

Class

\[ \Delta \]

behaviorFragment : \(\mathbb{P}\) BehaviorFragment

ownedServicedPort : \(\mathbb{P}\) ServicedPort ⋂

ownedPerceptor : \(\mathbb{P}\) Perceptor ⋂

ownedEffector : \(\mathbb{P}\) Effector ⋂

capability : \(\mathbb{P}\) Capability ⋂

BehavioredSemiEntityType = \(\emptyset\)

[1] \(\forall c:\text{capability}\bullet\)

\(\forall ob: self.\text{ownedBehavior}\bullet\)

\(\forall f : self.\text{feature} | \text{isKindOf}(f, \text{BehavioralFeature}) = \text{true}\bullet\)

\(c = ob \lor c = f\)

[2] \(\forall bf : \text{behaviorFragment}\bullet\)

\(\forall oa : self.\text{ownedAttribute} | \text{oa.aggregation} \in \{\text{shared, composite}\} \land \text{oa.type} \neq \emptyset \land \text{isKindOf}(oa, \text{BehaviorFragment}) = \text{true}\bullet\)

\(bf = \text{oa.type}\)

[3] \(\forall osp : \text{ownedServicedPort}\bullet\)

\(\forall op : self.\text{ownedPort} | \text{isKindOf}(op, \text{ServicedPort}) = \text{true}\bullet\)

\(osp = op\)

[4] \(\forall op : \text{ownedPerceptor}\bullet\)

\(\forall osp : self.\text{ownedServicedPort} | \text{isKindOf}(osp, \text{Perceptor}) = \text{true}\bullet\)

\(op = osp\)

[5] \(\forall oe : \text{ownedEffector}\bullet\)

\(\forall osp : self.\text{ownedServicedPort} | \text{isKindOf}(osp, \text{Effector}) = \text{true}\bullet\)

\(oe = osp\)

BehavioredSemiEntityType is an abstract Object-Z class that inherits from ServicedElement and Class classes. Following invariants are defined:

[1] The capability set is union of owned BehavioralFeatures and Behaviors.

[2] The behaviorFragment set comprises types of all owned aggregate or composite attributes having the type of a BehaviorFragment.

[3] The ownedServicedPort set refers to all owned ports of the kind ServicedPort.


[5] The ownedEffector set refers to all ownedServicePorts of the kind Effector.
9.1.2 Capability

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 [11]), 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 postconditions, 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. Capability is part of the non-conservative extension of UML, while it is a common superclass to two UML metaclasses: BehavioralFeature and Behavior. For more details see [1, p. 178].

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

Capability

\[(\ldots, \text{output}, \text{precondition}, \text{postcondition}, \text{input})\]

RedefinableElement

Namespace

\[\Delta\]

\begin{align*}
\text{output} & : \mathbb{P} \text{ Parameter} \land \\
\text{precondition} & : \mathbb{P} \text{ Constraint} \land \\
\text{postcondition} & : \mathbb{P} \text{ Constraint} \land \\
\text{input} & : \mathbb{P} \text{ Parameter} \land
\end{align*}

\[\text{Capability} = \emptyset\]

\[1\] \forall \ i : \text{input} \bullet
\begin{align*}
\text{if isKindOf(self, BehavioralFeature) = true} & \text{ then} \\
& \forall p : \text{asType(self, BehavioralFeature).parameter} \bullet \\
& \quad p.\text{direction} \in \{\text{in, inout}\}
\end{align*}

\[\text{else} \forall p : \text{asType(self, Behavior).parameter} \bullet
\begin{align*}
& \quad p.\text{direction} \in \{\text{in, out}\}
\end{align*}

\[\bullet \ i = p\]

\[2\] \forall o : \text{output} \bullet
\begin{align*}
\text{if isKindOf(self, BehavioralFeature) = true} & \text{ then} \\
& \forall p : \text{asType(self, BehavioralFeature).parameter} \bullet \\
& \quad p.\text{direction} \in \{\text{out, inout}\}
\end{align*}

\[\text{else} \forall p : \text{asType(self, Behavior).parameter} \bullet
\begin{align*}
& \quad p.\text{direction} \in \{\text{in, inout}\}
\end{align*}

\[\bullet \ o = p\]

\[3\] \forall p : \text{precondition} \bullet
\begin{align*}
\text{if isKindOf(self, Behavior) = true} & \text{ then} \\
& \forall \text{pre} : \text{asType(self, Behavior).precondition}
\end{align*}

\[\text{else} \]

\[\text{if isKindOf(self, Operation) = true} \text{ then} \\
& \forall \text{pre} : \text{asType(self, Operation).precondition}
\end{align*}

\[\text{else} \ \text{pre} = \emptyset \ (\text{self is the kind Reception})
\]

\[\bullet \ p = \text{pre}\]

\[4\] \forall p : \text{postcondition} \bullet
\begin{align*}
\text{if isKindOf(self, Behavior) = true} & \text{ then} \\
& \forall \text{pc} : \text{asType(self, Behavior).postcondition}
\end{align*}

\[\text{else} \]

\[\text{if isKindOf(self, Operation) = true} \text{ then} \\
& \forall \text{pc} : \text{asType(self, Operation).postcondition}
\end{align*}

\[\text{else} \ \text{pc} = \emptyset \ (\text{self is the kind Reception})
\]

\[\bullet \ p = \text{pc}\]

*Capability* is an abstract Object-Z class that inherits from *RedefinableElement* and *Namespace* classes. Following invariants are defined:
9.2 Behavior Decomposition

The Behavior Decomposition package defines the BehaviorFragment which allows the decomposition of complex behaviors of BehavioredSemiEntityTypes and the means to build reusable libraries of behaviors and related features.

9.2.1 BehaviorFragment

BehaviorFragment is a specialized BehavioredSemiEntityType 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 run time, the behaviored semi-entity delegates execution of its behavior to the containing BehaviorFragment instances. For more details see [1, p. 181].

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

BehaviorFragment

BehavioredSemiEntityType

BehaviorFragment class inherits from BehavioredSemiEntityType class.

9.3 Communicative Interactions

The Communicative Interactions package contains metaclasses that provide generic as well as agent specific extensions to UML Interactions. The generic extension allows the 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 allows the 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.

### 9.3.1 MultiLifeline

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. For more details see [1, p. 187].

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

![MultiLifeline Diagram]

MultiLifeline class inherits from Lifeline and MultiplicityElement classes.

### 9.3.2 MultiMessage

MultiMessage is a specialized Message (from UML) which is used to model a particular communication between MultiLifelines 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. For more details see [1, p. 189].
MultiMessage is introduced to model messages with multiple senders and/or recipients.

Object-Z class MultiMessage inherits from Message class. Following invariants must be satisfied:

1. At least one end of the MultiMessage must be a MultiLifeline.
2. The sendDiscriminator set can be specified only if the sender is represented by a MultiLifeline.
3. The receiveDiscriminator set can be specified only if the receiver is represented by a MultiLifeline.

9.3.3 DecoupledMessage

DecoupledMessage is a specialized MultiMessage which is used to model a specific kind of communication within an Interaction (from UML), particularly the asynchronous sending and receiving of a DecoupledMessagePayload 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. The objects transmitted in the form of DecoupledMessages are DecoupledMessagePayload instances. Because all the decoupled messages are asynchronous, the messageSort meta-attribute (inherited from the UML Message) is ignored. For more details see [1, p. 191].

DecoupledMessage is introduced to model autonomy in message processing.
9.3. Communicative Interactions

\[ \text{DecoupledMessage} \]
\[ \{ \ldots, \text{payload} \} \]
\[ \text{MultiMessage} \]

\[ \text{payload} : \mathbb{P} \text{DecoupledMessagePayload} \]
\[ \#\text{payload} \leq 1 \]

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

\textit{DecoupledMessage} class inherits from \textit{MultiMessage} class. Invariant [1] is expressed only in natural language due to absented UML metamodel.

9.3.4 \textbf{DecoupledMessagePayload}

\textit{DecoupledMessagePayload} is a specialized \textit{Class} (from UML) used to model the type of objects transmitted in the form of \textit{DecoupledMessages}. For more details see [1, p. 193]. 

\textit{DecoupledMessagePayload} is introduced to model objects transmitted in the form of \textit{DecoupledMessages}.

\[ \text{DecoupledMessagePayload} \]
\[ \text{Class} \]

\textit{DecoupledMessagePayload} class inherits from \textit{Class} class.

9.3.5 \textbf{Subset}

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

- an \textit{EventOccurrence} owned by the “superset” \textit{Lifeline} (client), and
- the “subset” \textit{Lifelines} (suppliers).

It is used to specify that since the occurrence of the \textit{supersetEvent}, some of the instances represented by the “superset” \textit{Lifeline} are also represented by the “subset” \textit{Lifeline}. The “subset” \textit{Lifeline}’s selector (for the details about the selector see \textit{Lifeline} [11] and section 9.3.1) specifies the instances of the “superset” \textit{Lifeline} that are also represented by the “subset” \textit{Lifeline}. All instances represented by the “subset” \textit{Lifeline} are still represented also by the “superset” \textit{Lifeline}. One \textit{Lifeline} can represent a “subset” of several “superset” \textit{Lifelines}, i.e. more than one \textit{Subset} relationships can lead to one “subset” \textit{Lifeline}. Termination of the “subset” \textit{Lifeline} (the \textit{Stop} is placed at the end of \textit{Lifeline}) destroys all instances it represents. For more details see [1, p. 194].

\textit{Subset} is introduced to specify that instances represented by one \textit{Lifeline} are a subset of instances represented by another \textit{Lifeline}. 
9.3. Communicative Interactions

Subset

\[(\ldots, \text{subset}, \text{supersetEvent})\]

Dependency

\[
\text{subset} : \mathbb{P} \text{ Lifeline} \\
\text{supersetEvent} : \mathbb{P} \text{ EventOccurrence}
\]

\[
\#\text{subset} \geq 1 \\
\#\text{supersetEvent} = 1 \\
[1] \quad \forall s : \text{subset} \mid \text{\textit{s}.represents.type} \neq \emptyset \Rightarrow \forall se : \text{supersetEvent} \mid \text{\textit{se}.covered.represents.type} \neq \emptyset \Rightarrow \\
\text{conformsTo}(\text{\textit{s}.present.type}, \text{\textit{se}.covered.represents.type}) = \text{true}
\]

Subset class inherits from Dependency class. Invariant [1] formalizes the fact that all types of the subset Lifelines must conform to the type of the superset Lifeline.

9.3.6 Join

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).

It is used to specify that a subset of instances, which have been until the \text{supersetEvent} represented by the “subset” Lifeline, is, after the \text{unionEvent} represented only by the “union” Lifeline. Thus after the \text{unionEvent} occurrence, the “union” Lifeline represents the union of the instances it has previously represented 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 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 \text{subsetEvent} and \text{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. For more details see [1, p. 196].

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

Join

\[(\ldots, \text{unionEvent}, \text{selector}, \text{subsetEvent})\]

Dependency

\[\text{unionEvent} : P \text{EventOccurrence}\]
\[\text{selector} : P \text{Expression} \circ\]
\[\text{subsetEvent} : P \text{EventOccurrence}\]

#unionEvent = 1
#selector \leq 1
#subsetEvent = 1

[1] \forall \text{ue} : \text{unionEvent} \bullet
   \text{iskindOf(ue.covered, MultiLifeline)} = \text{true}

[2] \forall \text{se} : \text{subsetEvent} | \text{se.covered.represents.type} \neq \emptyset \bullet
   \forall \text{ue} : \text{unionEvent} | \text{ue.covered.represents.type} \neq \emptyset \bullet
   \text{conformsTo(se.covered.represents.type, ue.covered.represents.type) = true}

Join class inherits from Dependency class. Following invariants must be satisfied:

[1] The Lifeline owning the EventOccurrence referred to by the unionEvent set must be a MultiLifeline.

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

9.3.7 AttributeChange

AttributeChange is a specialized InteractionFragment (from UML) used to model the change of attribute values (state) of the ConnectableElements (from UML) represented by Lifelines (from UML) within Interactions (from UML). AttributeChange enables to add, change or remove 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) 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. For more details see [1, p. 199].

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

\[ \text{AttributeChange} \]
\[ (\ldots, \text{destroyedLifeline}, \text{owningLifeline}, \text{when}, \text{createdLifeline}) \]
\[ \text{InteractionFragment} \]

\[ \text{destroyedLifeline} : \mathbb{P} \text{Lifeline} \]
\[ \text{owningLifeline} : \mathbb{P} \text{Lifeline} \]
\[ \text{when} : \mathbb{P} \text{EventOccurrence} \]
\[ \text{createdLifeline} : \mathbb{P} \text{Lifeline} \]

\[ \#\text{owningLifeline} \leq 0 \]
\[ \#\text{when} = 1 \]
\[ [1] \text{createdLifeline} \neq \emptyset \Rightarrow \text{owningLifeline} \neq \emptyset \]
\[ [2] \text{cl} : \text{createdLifeline} \mid \text{cl} \neq \emptyset \bullet \]
\[ \forall \text{ol} : \text{owningLifeline} \mid \text{ol}.\text{represents}.\text{type} \neq \emptyset \bullet \]
\[ \text{includesAll(ol}.\text{represents}.\text{attribute}, \text{cl}.\text{represents}) = \text{true} \]

\text{AttributeChange} class inherits from \text{InteractionFragment} class. Invariant [1] says that if \text{createdLifeline} is specified, the \text{owningLifeline} must be specified as well. Invariant [2] specifies the fact that each \text{createdLifeline} must represent an \text{attribute} of the \text{Classifier} used as the \text{type} of the \text{ConnectableElement} represented by the \text{owningLifeline} set.

9.3.8 CommunicationSpecifier

\text{CommunicationSpecifier} is an abstract metaclass which defines metaproperties of its concrete subclasses, i.e. \text{CommunicationMessage}, \text{CommunicativeInteraction}, and \text{ServiceSpecification}, which are used to model different aspects of communicative interactions. \text{CommunicationMessages} can occur in \text{CommunicativeInteractions}, and parameterized \text{CommunicativeInteractions} can be parts of \text{ServiceSpecifications}. All of them can specify values of the meta-attributes inherited from the \text{CommunicationSpecifier}. Potential conflicts in specifications of the \text{CommunicationSpecifier}’s meta-property values are resolved by the overriding principle that defines which concrete subclasses of the \text{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. \text{CommunicationMessage},
2. \text{CommunicativeInteraction},
3. \text{ServiceSpecification}.

For more details see [1, p. 203].

\text{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.
9.3. Communicative Interactions

\[
\text{CommunicationSpecifier} \doteq (\text{acl, cl, encoding, ontology})
\]

\[
\begin{align*}
\text{acl} & : \text{seq ValueSpecification} \\
\text{cl} & : \text{seq ValueSpecification} \\
\text{encoding} & : \text{seq ValueSpecification} \\
\text{ontology} & : \text{seq ValueSpecification}
\end{align*}
\]

\[
\text{CommunicationSpecifier} = \emptyset \\
\#\text{acl} \leq 1 \\
\#\text{cl} \leq 1 \\
\#\text{encoding} \leq 1
\]

\text{CommunicationSpecifier} is an abstract Object-Z class.

9.3.9 CommunicationMessage

\text{CommunicationMessage} is a specialized \text{DecoupledMessage} and \text{CommunicationSpecifier}, which is used to model communicative acts of \text{speech act based communication} in the context of \text{Interactions}. The objects transmitted in the form of \text{CommunicationMessages} are \text{CommunicationMessagePayload} instances. For more details see [1, p. 204].

\text{CommunicationMessage} is introduced to model speech act based communication in the context of \text{Interactions}.

\[
\text{CommunicationMessage} \doteq (\ldots, \text{payload})
\]

\[
\begin{align*}
\text{DecoupledMessage} \\
\text{CommunicationSpecifier}
\end{align*}
\]

\[
\text{payload} : \mathbb{P} \text{CommunicationMessagePayload}
\]

\[
\#\text{payload} \leq 1
\]

Object-Z class \text{CommunicationMessage} inherits from \text{DecoupledMessage} and \text{CommunicationSpecifier} classes.

9.3.10 CommunicationMessagePayload

We introduce a \text{String} data type representing the set of all possible sequences of characters (this is a given type in Object-Z [18]).

\[
\text{[String]}
\]

\text{CommunicationMessagePayload} is a specialized \text{Class} (from UML) used to model the type of objects transmitted in the form of \text{CommunicationMessages}.

\text{CommunicationMessagePayload} is introduced to model objects transmitted in the form of \text{CommunicationMessages}. 
9.3. Communicative Interactions

CommunicationMessagePayload class inherits from DecoupledMessagePayload class.

9.3.11 CommunicativeInteraction

CommunicativeInteraction is a specialized Interaction (from UML) and CommunicationSpecifier, used to model speech act based communications, i.e. Interactions containing CommunicationMessages. 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 9.3.8 holds. For more details see [1, p. 207].

CommunicativeInteraction is introduced to model speech act based communications.

CommunicativeInteraction class inherits from Interaction and CommunicationSpecifier classes.

9.3.12 InteractionProtocol

InteractionProtocol is a parameterized CommunicativeInteraction 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,
9.3. Communicative Interactions

- \textit{MultiLifeline} multiplicities,
- \textit{MultiMessage} discriminators,
- \textit{CommunicationMessage} meta-attributes,
- \textit{ExecutionOccurrence}’s behavior specification,
- guard expressions of \textit{InteractionOperands},
- specification of included \textit{Constraints}, and
- included \textit{Expressions} and their particular operands.

Partial binding of an \textit{InteractionProtocol} (i.e. the \textit{TemplateBinding} which does not substitute all the template parameters by actual parameters) results in a different \textit{InteractionProtocol}. A complete binding of an \textit{InteractionProtocol} represents a \textit{CommunicativeInteraction}. For more details see [1, p. 208].

\textit{InteractionProtocol} is introduced to model reusable templates of \textit{CommunicativeInteractions}.

\begin{center}
\begin{tabular}{l}
\textbf{InteractionProtocol} \\
\hline
| (. . . , \textit{ownedSignature}) \\
\textit{CommunicativeInteraction} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{l}
\textit{ownedSignature} : \mathbb{P} \textit{RedefinableTemplateSignature} \\
\texttt{#ownedSignature} = 1
\end{tabular}
\end{center}

\textit{InteractionProtocol} class inherits from \textit{CommunicativeInteraction} class.

### 9.3.13 SendDecoupledMessageAction

\textit{SendDecoupledMessageAction} is a specialized \textit{SendObjectAction} (from UML) used to model the action of sending of \textit{DecoupledMessagePayload} instances, referred to by the \textit{request} meta-association, in the form of a \textit{DecoupledMessage} to its recipient(s), referred to by the \textit{target} meta-association. For more details see [1, p. 213].

\textit{SendDecoupledMessageAction} is introduced to model the sending of \textit{DecoupledMessages} in \textit{Activities}.

\begin{center}
\begin{tabular}{l}
\textbf{SendDecoupledMessageAction} \\
\hline
| (. . . , \textit{target}) \\
\textit{SendObjectAction} \\
\hline
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{l}
\textit{target} : \mathbb{P} \textit{InputPin} \\
\texttt{#target} \geq 1
\end{tabular}
\end{center}

\textit{SendDecoupledMessageAction} class inherits from \textit{SendObjectAction} class.
9.3. Communicative Interactions

9.3.14 SendCommunicationMessageAction

SendCommunicationMessageAction is a specialized SendDecoupledMessageAction, which allows to specify the values of the CommunicationSpecifier’s meta-attributes. For more details see [1, p. 214].

SendCommunicationMessageAction is introduced to model the sending of Communication-Messages in Activities.

\[ \text{SendCommunicationMessageAction} \]
\[ \text{SendDecoupledMessageAction} \]
\[ \text{CommunicationSpecifier} \]

SendCommunicationMessageAction is an Object-Z class, which inherits from SendDecoupledMessageAction and CommunicationSpecifier classes.

9.3.15 AcceptDecoupledMessageAction

AcceptDecoupledMessageAction is a specialized AcceptEventAction (from UML) which waits for the reception of a DecoupledMessage that meets conditions specified by the associated trigger (for details see section 9.3.17). The received DecoupledMessagePayload 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. For more details see [1, p. 217].

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

\[ \text{AcceptDecoupledMessageAction} \]
\[ \langle \ldots, \text{result}, \text{trigger} \rangle \]
\[ \text{AcceptEventAction} \]

\[ \text{result} : \mathbb{P} \text{OutputPin} \odot \]
\[ \text{trigger} : \mathbb{P} \text{DecoupledMessageTrigger} \]

\[
\#\text{result} = 1 \\
\#\text{trigger} = 1 \\
[1] \forall r : \text{result} \mid r.\text{type} \neq \emptyset \bullet \\
\text{isKindOf}(r.\text{type}, \text{DecoupledMessagePayload}) = \text{true}
\]

AcceptDecoupledMessageAction class inherits from AcceptEventAction class. Invariant [1] specifies following condition – if the type of the OutputPin referred to by the result set is specified, it must be a DecoupledMessagePayload.
9.3.16 AcceptCommunicationMessageAction

AcceptCommunicationMessageAction is a specialized AcceptEventAction (from UML) which waits for the reception of a CommunicationMessage that meets conditions specified by associated trigger (for details see section 9.3.18). The received CommunicationMessagePayload 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. For more details see [1, p. 218].

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

\[
\text{AcceptCommunicationMessageAction} \quad \ni \quad (\ldots, \text{result}, \text{trigger}) \\
\text{AcceptEventAction}
\]

\[
\begin{align*}
\text{result} & : \mathbb{P} \text{ OutputPin} \\
\text{trigger} & : \mathbb{P} \text{ CommunicationMessageTrigger}
\end{align*}
\]

\[
\begin{align*}
\#\text{result} &= 1 \\
\#\text{trigger} &= 1 \\
[1] & \forall r : \text{result} \mid r.\text{type} \neq \emptyset \bullet \\
& \quad \text{isKindOf}(r.\text{type}, \text{CommunicationMessagePayload}) = \text{true}
\end{align*}
\]

AcceptCommunicationMessageAction class inherits from AcceptEventAction. Invariant [1] says that if the type of the OutputPin referred to by the result set is specified, it must be a CommunicationMessagePayload.

9.3.17 DecoupledMessageTrigger

DecoupledMessageTrigger is a specialized Trigger (from UML) that represents the event of reception of a DecoupledMessage, 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 instance. For more details see [1, p. 219].

DecoupledMessageTrigger is introduced to model events representing reception of DecoupledMessages.
9.4 Services

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

9.4.1 ServiceSpecification

ServiceSpecification is a specialized BehavioredClassifier (from UML) and CommunicationSpecifier, used to specify services. A service is a coherent block of functionality provided by a behaviored semi-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, i.e. InteractionProtocols 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 (clientParameter 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 section 9.3.8 applies. For more details see [1, p. 223].

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.

\[
\text{ServiceSpecification} \sqsubseteq \text{BehavioredClassifier}\big/\Delta \text{CommunicationSpecifier} \\
| \ldots, \text{serviceProtocol} \\
\text{serviceProtocol} : \mathbb{P} \text{ServiceProtocol} \oplus \\
\# \text{serviceProtocol} \geq 1
\]

ServiceSpecification class inherits from ServiceSpecification class.

### 9.4.2 ServiceProtocol

ServiceProtocol is a specialized InteractionProtocol, used only within the context of its owning ServiceSpecification, 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 parameterized
CommunicativeInteraction) or as a partially bound, already defined InteractionProtocol. For more details see [1, p. 225].

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.

\[
\begin{align*}
&\text{ServiceProtocol} \\
&\left[\ldots, \text{providerParameter}, \text{clientParameter}\right] \\
&\text{InteractionProtocol}
\end{align*}
\]

\[
\begin{align*}
\text{providerParameter} & : \mathbb{P} \text{TemplateParameter} \\
\text{clientParameter} & : \mathbb{P} \text{TemplateParameter} \\
\#\text{providerParameter} & \geq 1 \\
\#\text{clientParameter} & \geq 1 \\
[1] & \forall p : \text{self.ownedSignature.parameter} \bullet \\
&\forall pp : \text{providerParameter} \bullet \\
&\text{includesAll}(p, pp) = \text{true} \\
[2] & \forall p : \text{self.ownedSignature.parameter} \bullet \\
&\forall cp : \text{clientParameter} \bullet \\
&\text{includesAll}(p, cp) = \text{true} \\
[3] & \text{providerParameter} \cap \text{clientParameter} = \emptyset \\
[4] & \text{providerParameter} \cup \text{clientParameter} = \text{self.ownedSignature.parameter}
\end{align*}
\]

ServiceProtocol class inherits from InteractionProtocol class. Following invariants must be satisfied:

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

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


9.4.3 ServicedElement

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. BehavioralSemiEntity, ServicedPort, and ServicedProperty). Technically, the service provision and usage is modeled by ownership of ServiceProvisions and ServiceUsages. For more details see [1, p. 228].

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

ServicedElement

\[(\ldots, \text{serviceProvision, serviceUsage})\]

NamedElement

\[\Delta\]

\[\text{serviceProvision : } \mathbb{P} \text{ ServiceProvision} \]

\[\text{serviceUsage : } \mathbb{P} \text{ ServiceUsage} \]

ServicedElement = \emptyset

\[\forall \ o : \text{serviceProvision} \bullet \text{self} = o.\text{provider} \]

\[\forall \ o : \text{serviceUsage} \bullet \text{self} = o.\text{client} \]

[1] \[\forall \ sp : \text{serviceProvision} \bullet\]

\[\forall \ cd : \text{self}.\text{clientDependency} \mid \text{isKindOf}(cd, \text{ServiceProvision}) = \text{true} \bullet\]

\[\text{sp} = cd \]

[2] \[\forall \ su : \text{serviceUsage} \bullet\]

\[\forall \ cd : \text{self}.\text{clientDependency} \mid \text{isKindOf}(cd, \text{ServiceUsage}) = \text{true} \bullet\]

\[\text{su} = cd \]

ServicedElement is an abstract class that inherits from NamedElement. Invariant [1] says that the serviceProvision set refers to all clientDependencies of the kind ServiceProvision. Invariant [2] states that the serviceUsage set refers to all clientDependencies of the kind ServiceUsage.

9.4.4 ServicedProperty

ServicedProperty is a specialized Property (from UML) and ServicedElement, 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 and ServiceUsages owned by the the ServicedProperty are handled by its type. For details see section 9.1.1. For more details see [1, p. 229].

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

\[\text{ServicedProperty} \]

\[\mid (\ldots, \text{type})\]

Property

ServicedElement

\[\text{type} : \mathbb{P} \text{ BehavioredSemiEntityType} \]

\[#\text{type} \leq 1 \]

ServicedProperty class inherits from Property and ServicedElement classes.
9.4.5 ServicedPort

ServicedPort is a specialized Port (from UML) and ServicedElement that specifies a distinct interaction point between the owning BehavioredSemiEntityType 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. 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 and ServiceUsages owned by the the ServicedPort are handled by its type. For details see section 9.1.1 For more details see [1, p. 231].

\[
\text{ServicedPort} \leadsto (\ldots, \text{type}) \\
\text{Port} \\
\text{ServicedElement} \\
\text{type} : \mathbb{P} \text{BehavioredSemiEntityType} \\
\#\text{type} \leq 1
\]

ServicedPort class inherits from Port and ServicedElement classes.

9.4.6 ServiceProvision

ServiceProvision is a specialized Realization dependency (from UML) between a ServiceSpecification and a ServicedElement, 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, which are partially bound counterparts to all ServiceProtocols 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 9.4.7) 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 according to which the service is accessed. For more details see [1, p. 233].

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

ServiceProvision class inherits from Realization class. Invariant [1] formalizes the fact that the providingIP binds all (and only) the providerParameters from all the service’s ServiceProtocols.

9.4.7 ServiceUsage

ServiceUsage is a specialized Usage dependency (from UML) between a ServiceSpecification and a ServicedElement, 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, which are partially bound counterparts to all ServiceProtocols 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 9.4.6) 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 according to which the service is accessed. For more details see [1, p. 235].

ServiceUsage is introduced to specify that the ServicedElement uses the service specified by the related ServiceSpecification.
9.5 Observations and Effecting Interactions

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).

### 9.5.1 PerceivingAct

*PerceivingAct* is a specialized *Operation* (from UML) which is owned by a *PerceptorType* and thus can be used to specify what perceptions the owning *PerceptorType*, or a *Perceptor* of that *PerceptorType*, can perform. For more details see [1, p. 238].

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

---

ServiceUsage class inherits from Usage class. Invariant [1] says that the usageIP binds all and only the clientParameters from all service’s ServiceProtocols.
9.5.2 PerceptorType

PerceptorType is a specialized BehavioredSemiEntityType used to model the type of Perceptors, in terms of owned:

- Receptions (from UML) and
- PerceivingActs.

For more details see [1, p. 239]. PerceptorType is introduced to model types of Perceptors.

PerceptorType class inherits from BehavioredSemiEntityType class. Invariant [1] says that the ownedPerceivingAct set refers to all ownedOperations of the kind PerceivingAct.

9.5.3 Perceptor

Perceptor is a specialized ServicedPort used to model capability of its owner (a BehavioredSemiEntityType) 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. For more details see [1, p. 241]. Perceptor is introduced to model the capability of its owner to observe.

Perceptor class inherits from ServicedPort class.
9.5.4 PerceptAction

PerceptAction is a specialized CallOperationAction (from UML) which can call PerceivingActs. 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. For more details see [1, p. 243].

PerceptAction is introduced to model observations in Activities.

```
PerceptAction
| (... , perceivingAct) |
CallOperationAction

perceivingAct : \( \mathbb{P} \) PerceivingAct

#perceivingAct = 1
```

PerceptAction class inherits from CallOperationAction class.

9.5.5 Perceives

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,
- PerceivingActs,
- PerceptorTypes, or
- Perceptors.

For more details see [1, p. 244].

Perceives is introduced to model which elements can observe others.

```
Perceives
Dependency
```

Perceives class inherits from Dependency class.

9.5.6 EffectingAct

EffectingAct is a specialized Operation (from UML) which is owned by an EffectorType and thus can be used to specify what effecting acts the owning EffectorType, or an Effector of that EffectorType, can perform. For more details see [1, p. 245].
**EffectingAct** is introduced to specify which effecting acts the owning **EffectorType**, or an **Effector** of that **EffectorType**, can perform.

\[
\text{EffectingAct}
\] 
\[
| (\ldots, \text{effectorType})
\] 
\[
\text{Operation}
\]

\[
\text{effectorType} : \mathbb{P} \text{EffectorType}
\]
\[
\#\text{effectorType} = 1
\]
\[
\forall o : \text{effectorType} \bullet self \in o.\text{ownedEffectingAct}
\]

**EffectingAct** is an Object-Z class, which inherits from **Operation**.

### 9.5.7 **EffectorType**

**EffectorType** is a specialized **BehavioredSemiEntityType** used to model type of **Effectors**, in terms of owned **EffectingActs**. For more details see [1, p. 246].

**EffectorType** is introduced to model types of **Effectors**.

\[
\text{EffectorType}
\] 
\[
| (\ldots, \text{ownedEffectingAct})
\] 
\[
\text{BehavioredSemiEntityType}
\]

\[
\Delta
\]
\[
\text{ownedEffectingAct} : \mathbb{P} \text{EffectingAct} \odot
\]
\[
\forall o : \text{ownedEffectingAct} \bullet o.\text{effectorType} = \text{self}
\]
\[
[1] \forall oea : \text{ownedEffectingAct} \bullet
\]
\[
\forall oo : self.\text{ownedOperation} | \text{isKindOf}(oo, \text{EffectingAct}) = \text{true} \bullet
\]
\[
oea = oo
\]

**EffectorType** class inherits from **BehavioredSemiEntityType** class. Invariant [1] formalizes the fact that the **ownedEffectingAct** set refers to all **ownedOperations** of the kind **EffectingAct**.

### 9.5.8 **Effector**

**Effector** is a specialized **ServicedPort** used to model the capability of its owner (a **BehavioredSemiEntityType**) 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**. For more details see [1, p. 247].

**Effector** is introduced to model the capability of its owner to bring about an effect on other objects.
9.5. Observations and Effecting Interactions

**Effector** class inherits from **ServicedPort** class.

### 9.5.9 EffectAction

**EffectAction** is a specialized **CallOperationAction** (from UML) which can call **EffectingActs**. 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. For more details see [1, p. 248].

**EffectAction** is introduced to model effections in **Activities**.

**EffectAction** class inherits from **CallOperationAction** class.

### 9.5.10 Effects

**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**,
- **EffectingActs**,
- **EffectorTypes**, or
- **Effectors**.

For more details see [1, p. 249].

**Effects** is introduced to model which elements can effect others.
9.6 Mobility

The Mobility package defines metaclasses used to model structural and behavioral aspects of entity mobility.

9.6.1 Move

Move is a specialized Dependency (from UML) between two HostingProperties used to specify that the entities represented by the source HostingProperty (specified by the from meta-association) can be moved/transfered to the instances of the AgentExecutionEnvironment owning the destination HostingProperty (specified by the to meta-association). For more details see [1, p. 250].

Move is introduced to model the movement of entities between instances of AgentExecutionEnvironments.

\[ \text{Move} \]
\[ \{(\ldots, \text{from}, \text{to})\} \]
\[ \text{Dependency} \]

\[
\begin{align*}
\text{from} &: \mathbb{P} \text{HostingProperty} \\
\text{to} &: \mathbb{P} \text{HostingProperty} \\
#\text{from} &= 1 \\
\forall o : \text{from} &\cdot \text{self} \in o.\text{move} \\
#\text{to} &= 1 \\
\forall o : \text{to} &\cdot \text{self} \in o.\text{moveFrom} \\
[1] \forall f : \text{from} &\mid f.\text{type} \neq \emptyset \\
\forall t : \text{to} &\mid t.\text{type} \neq \emptyset \\
&\text{conformsTo}(t.\text{type}, f.\text{type})
\end{align*}
\]

Move class inherits from Dependency class. Invariant [1] says that if both types are specified, then the type of the HostingProperty referred to by the to set must conform to the type of the HostingProperty referred to by the from set.

9.6.2 Clone

Clone is a specialized Dependency (from UML) between HostingProperties 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 owning the destination HostingProperties (specified by the to meta-association). For more details see [1, p. 252].
 Clone is introduced to model the cloning of entities among instances of AgentExecution-Environments.

\[
\begin{align*}
\text{Clone} &\; \mid (\ldots, \text{from}, \text{to}) \\
\text{Dependency} &
\end{align*}
\]

\[
\begin{align*}
\text{from} &\; : \mathbb{P} \text{HostingProperty} \\
\text{to} &\; : \mathbb{P} \text{HostingProperty} \\
\#\text{from} &\; = 1 \\
\forall o : \text{from} &\; \bullet \text{self} \in o.\text{clone} \\
\#\text{to} &\; \geq 1 \\
\forall o : \text{to} &\; \bullet \text{self} \in o.\text{clone} \\
[1] \forall f : \text{from} &\; \mid f.\text{type} \neq \emptyset \bullet \\
\forall t : \text{to} &\; \mid t.\text{type} \neq \emptyset \bullet \\
&\; \text{conformsTo}(t.\text{type}, f.\text{type})
\end{align*}
\]

Clone class inherits from Dependency class. Invariant [1] express the following condition – if both types are specified, then the types of the HostingProperties referred to by the to set must conform to the type of the HostingProperty referred to by the from set.

### 9.6.3 MobilityAction

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 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 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 and
- CloneAction.

For more details see [1, p. 253].

MobilityAction is introduced to define the common features of all its subclasses.
9.6. Mobility

MobilityAction is an abstract class that inherits from AddStructuralFeatureValueAction class. Following invariants must be satisfied:

[1] If the type of the InputPin referred to by the entity set is specified, it must be an EntityType.

[2] If the type of the InputPin referred to by the to set is specified, it must be an AgentExecutionEnvironment.

[3] If the type of the InputPin referred to by the to set is specified, the HostingProperty referred to by the toHostingProperty set must be an ownedAttribute of that type.

9.6.4 MoveAction

MoveAction is a specialized MobilityAction 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 instance (specified as the to meta-association, inherited from MobilityAction). For more details see [1, p. 255].

MoveAction is introduced to model the movement of entities in Activities.

MoveAction class inherits from MobilityAction class.
9.6. Mobility

9.6.5 CloneAction

CloneAction is a specialized MobilityAction used to model an action that results in an 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 instance (specified as the to meta-association, inherited from MobilityAction). 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). For more details see [1, p. 256].

CloneAction is introduced to model the cloning of entities in Activities.

\[
\text{CloneAction} \leftarrow (\ldots, \text{clone}) \\
\text{MobilityAction}
\]

\[
\text{clone} : \mathcal{P} \text{OutputPin} \odot
\]

\[
\#\text{clone} = 1
\]

\[1 \quad \forall c : \text{clone} \mid c\.\text{type} \neq \emptyset \bullet
\]

\[
is\text{KindOf}(c\.\text{type}, \text{EntityType}) = \text{true}
\]

\[2 \quad \forall c : \text{clone} \mid c\.\text{type} \neq \emptyset \bullet
\]

\[
\forall e : \text{self.entity} \mid e\.\text{type} \neq \emptyset \\
\text{conformsTo}(c\.\text{type}, e\.\text{type}) = \text{true}
\]

CloneAction class inherits from MobilityAction class. Following invariants must be satisfied:

[1] If the type of the OutputPin referred to by the clone set is specified, it must be an EntityType.

[2] The type of the OutputPin referred to by the clone set must conform to the type of the InputPin referred to by the entity set, if the both specified.
Chapter 10

Mental

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.

10.1 Mental States

The Mental States package comprises common fundamental metaclasses used to define concrete metaclasses contained within the rest of the Mental sub-packages, i.e. Beliefs, Goals, Plans and Mental Relationships.

10.1.1 MentalState

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, 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.
10.1. Mental States

For more details see [1, p. 263].

*MentalState* is introduced to define the common features of all its subclasses.

\[
\begin{align*}
\text{MentalState} \\
\{(. . ., \text{degree})
\end{align*}
\]

\[
\begin{align*}
\text{NamedElement} \\
\Delta \\
\text{degree} : \text{seq} \text{ValueSpecification} \\
\text{MentalState} = \emptyset \\
\#\text{degree} \leq 1
\end{align*}
\]

*MentalState* class inherits from *NamedElement* class.

10.1.2 MentalClass

*MentalClass* is an abstract specialized *Class* (from UML) and *MentalState* serving as a common superclass to all the metaclasses which can be used to specify mental attitudes of *MentalSemiEntityTypes*. Technically, *MentalProperties* can only be of the *MentalClass* type. Furthermore, the object meta-association of the *Responsibility* relationship can also only be of the *MentalClass* type. For more details see [1, p. 264].

*MentalClass* is introduced to specify the mental attitudes of *MentalSemiEntityTypes* and objects of *Responsibility* relationship.

\[
\begin{align*}
\text{MentalClass} \\
\{(. . ., \text{isResponsibilityOf})
\end{align*}
\]

\[
\begin{align*}
\text{Class} \\
\text{MentalState} \\
\Delta \\
\text{isResponsibilityOf} : \mathbb{P} \text{Responsibility} \\
\text{MentalClass} = \emptyset \\
\forall o : \text{isResponsibilityOf} \cdot \text{self} \in o.\text{object} \\
[1] \forall iro : \text{isResponsibilityOf} \cdot \\
\forall sd : \text{self.supplierDependency} | \text{isKindOf}(sd, \text{Responsibility}) = \text{true} \cdot \\
\text{iro} = sd
\end{align*}
\]

*MentalClass* is an abstract class that inherits from *Class* and *MentalState* classes. Invariant [1] says that the *isResponsibilityOf* set refers to all *supplierDependencies* of the kind *Responsibility*.

10.1.3 ConstrainedMentalClass

*ConstrainedMentalClass* is an abstract specialized *MentalClass* which allows its concrete subclasses to specify *MentalConstraints*. 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. For more details see [1, p. 264].

ConstrainedMentalClass is introduced to allow the specification of MentalConstraints for all its subclasses.

\[
\begin{align*}
\text{ConstrainedMentalClass} & \subseteq \text{MentalClass} \\
\Delta \\
\text{mentalConstraint} : \exists \text{MentalConstraint} \\
\text{ConstrainedMentalClass} = \emptyset \\
[1] & \forall mc_1, mc_2: \text{mentalConstraint} \bullet \\
& mc_1.\text{kind} \neq mc_2.\text{kind} \\
[2] & \forall mc: \text{mentalConstraint} \bullet \\
& \forall \text{or} : \text{self.ownedRule} \mid \text{isKindOf}(\text{or, MentalConstraint}) = \text{true} \bullet \\
& mc = \text{or}
\end{align*}
\]

ConstrainedMentalClass is an abstract class that inherits from MentalClass class. Following invariants must be satisfied:

[1] Each mentalConstraint must have a different kind.


### 10.1.4 MentalConstraint

MentalConstraint is a specialized Constraint (from UML) and RedefinableElement (from UML), used to specify properties of ConstrainedMentalClasses 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 10.1.5). 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 Mental-Properties. For more details see [1, p. 265].

MentalConstraint is introduced to specify the properties of ConstrainedMentalClasses which can be used within mental (reasoning) processes of owning MentalSemiEntityTypes.
MentalConstraint class inherits from RedefinableElement and Constraint classes. Invariant [1] says that the commitPreCondition literal cannot be the value of the kind object.

### 10.1.5 MentalConstraintKind

MentalConstraintKind is an enumeration which specifies kinds of MentalConstraints, as well as kinds of constraints specified for contributor and beneficiary in the Contribution relationship. If needed, the set of MentalConstraintKind enumeration literals can be extended. For more details see [1, p. 266].

MentalConstraintKind is introduced to specify the kinds of MentalConstraints and ends of a Contribution relationship.

\[
\text{MentalConstraintKind} ::= \text{commitCondition} | \text{preCondition} | \text{commitPreCondition} \\
| \text{invariant} | \text{cancelCondition} | \text{postCondition}
\]

MentalConstraintKind is an enumeration, which has commitCondition, preCondition, commitPreCondition, invariant, cancelCondition, and postCondition values.

### 10.1.6 MentalRelationship

MentalRelationship is an abstract specialized MentalState, a superclass to all metaclasses defining the relationships between MentalStates. There is one concrete subclass of the MentalRelationship–Contribution. For more details see [1, p. 267].

MentalRelationship is introduced as a superclass to all metaclasses defining the relationships between MentalStates.

\[
\text{MentalRelationship} \subseteq \text{MentalState}
\]

MentalRelationship is an abstract class, which inherits from MentalRelationship class.

### 10.1.7 MentalSemiEntityType

MentalSemiEntityType is a specialized abstract Class (from UML), a superclass to all metaclasses which can own MentalProperties, i.e. AutonomousEntityType and EntityRole-
Type. The ownership of a MentalProperty of a particular MentalClass 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 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 to execute, etc.

Instances of MentalSemiEntityType are referred to as mental semi-entities. For more details see [1, p. 268].

MentalSemiEntityType is introduced as a common superclass to all metaclasses which can own MentalProperties.

MentalSemiEntityType\(\upharpoonright (\ldots, \text{mentalAttribute})\)

\[\Delta\]

\[
\text{mentalAttribute} : \mathbb{P} \text{MentalProperty}
\]

\[\text{MentalSemiEntityType} = \emptyset\]

\[\forall \text{ma} : \text{mentalAttribute} \bullet \forall \text{oa} : \text{self.ownedAttribute} | \text{isKindOf}(\text{oa}, \text{MentalProperty}) = \text{true} \bullet \text{ma} = \text{oa}\]

MentalSemiEntityType is an abstract class, which inherits from Class class. Invariant [1] express following condition – the mentalAttribute set refers to all ownedAttributes of the kind MentalProperty.

10.1.8 MentalProperty

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 of its type, e.g. can decide whether to believe or not (and to what extent) in a Belief, or whether and when to commit to a Goal. 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 as attributes, or
- MentalAssociations as member ends.
MentalProperties (except of MentalProperties of Belief type) can own MentalConstraints (each of a different type) to allow the redefinition of MentalConstraints of their types. The redefinition rules are described in section 10.1.4. Note: The Plans controlled by MentalSemiEntityTypeTypes are modeled as owned UML Activities, and therefore use of Plans as types of MentalProperties is forbidden, even if they are specialized MentalClasses. For more details see [1, p. 268].

MentalProperty is introduced to specify that mental semi-entities have control over Goal and Belief instances.

$$\text{MentalProperty} \ \\
\{(..., \text{degree, association, type, mentalConstraint})\}$$

$$\text{Property}$$

$$\begin{align*}
\text{degree} &: \text{seq ValueSpecification} \\
\text{association} &: \mathbb{P} \text{MentalAssociation} \\
\text{type} &: \mathbb{P} \text{MentalClass} \\
\text{mentalConstraint} &: \mathbb{P} \text{MentalConstraint} \\
\#\text{degree} &\leq 1 \\
\#\text{association} &\leq 1 \\
\forall o : \text{association} \bullet o.\text{mentalMemberEnd} = \text{self} \\
\#\text{type} &\leq 1 \\
[1] \text{self.type} \neq \emptyset \Rightarrow \text{isKindOf(self.type, Plan)} = \text{false} \\
[2] \forall mc1, mc2 : \text{mentalConstraint} \bullet \\
&mcl.\text{kind} \neq mc2.\text{kind} \\
[3] \neg (\text{self.type} \neq \emptyset \land \text{isKindOf(self.type, ConstrainedMentalClass = true)}) \\
\Rightarrow \text{mentalConstraint} = \emptyset
\end{align*}$$

MentalProperty class inherits from Property class. Following invariants must be satisfied:

[1] If the type set is specified, the MentalClass referred to by it cannot be a Plan.

[2] Each mentalConstraint must have different kind.

[3] The mentalConstraints can be specified only for a ConstrainedMentalClass.

10.1.9 MentalAssociation

MentalAssociation is a specialized Association (from UML) between a MentalSemiEntityType and a MentalClass used to specify a MentalProperty of the MentalSemiEntityType in the form of an association end. MentalAssociation is always binary. An instance of the MentalAssociation is called mental link. For more details see [1, p. 272].

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.
10.2. Beliefs

**Beliefs**

The **Beliefs** package defines metaclasses used to model beliefs.

10.2.1 Belief

**Belief** is a specialized **MentalClass** 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 semientity’s **MentalProperty**, it represents the information which
the mental semi-entity believes, and which does not need to be objectively true. The ability of a MentalSemiEntity 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. For more details see [1, p. 275].

Belief is introduced to model beliefs.

<table>
<thead>
<tr>
<th>Belief</th>
<th>MentalClass</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Belief class inherits from MentalClass class.

10.3 Goals

The Goals package defines metaclasses used to model goals.

10.3.1 Goal

Goal is an abstract specialized ConstrainedMentalClass 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. 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, or max-Price. 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. For more details see [1, p. 279].
10.4. Plans

"Goal" is introduced to define the common features of all its subclasses that are used to model concrete types of goals.

\[
\begin{align*}
\text{Goal} & \triangleright \text{(..., degree)} \\
\text{ConstrainedMentalClass} & \\
\text{degree} : \text{seq ValueSpecification} & \\
\text{Goal} = \emptyset & \\
\# \text{degree} \leq 1 & 
\end{align*}
\]

"Goal" is an abstract class that inherits from "ConstrainedMentalClass" class.

10.3.2 DecidableGoal

"DecidableGoal" is a specialized concrete "Goal" used to model goals for which there are clear-cut criteria according to which the goal-holder can decide whether the "DecidableGoal" (particularly its postCondition; for details see section 10.1.5) has been achieved or not. For more details see [1, p. 278].

"DecidableGoal" is introduced to explicitly model decidable goals.

\[
\begin{align*}
\text{DecidableGoal} & \quad \text{Goal}
\end{align*}
\]

"DecidableGoal" class inherits from "Goal" class.

10.3.3 UndecidableGoal

"UndecidableGoal" is a specialized concrete "Goal" used to model goals for which there are no clear-cut criteria according to which the goalholder can decide whether the postCondition (see section 10.1.5 for details) of the "UndecidableGoal" is achieved or not. For more details see [1, p. 280].

"UndecidableGoal" is introduced to explicitly model undecidable goals.

\[
\begin{align*}
\text{UndecidableGoal} & \quad \text{Goal}
\end{align*}
\]

"UndecidableGoal" class inherits from "Goal" class.

10.4 Plans

The "Plans" package defines metaclasses devoted to modeling plans.
10.4. Plans

10.4.1 Plan

Plan is a specialized ConstrainedMentalClass and Activity (from UML), used to model capabilities of MentalSemiEntityTypes which represents either:

- predefined plans, i.e. kinds of activities a mental semi-entity’s reasoning mechanism can manipulate in order to achieve Goals, 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 10.1.5), which can be used by reasoning mechanisms. For modeling the applicability of Plans, in relation to given Goals, Beliefs and other Plans, the Contribution relationship 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. For more details see [1, p. 282].

Plan is introduced to model predefined plans, or fragments of plans from which the plans can be composed.

<table>
<thead>
<tr>
<th>Plan</th>
</tr>
</thead>
<tbody>
<tr>
<td>(..., degree)</td>
</tr>
<tr>
<td>Activity</td>
</tr>
<tr>
<td>ConstrainedMentalClass</td>
</tr>
</tbody>
</table>

\[
degree : \text{seq ValueSpecification}
\]

\[
\text{#} \degree \leq 1
\]

\[1\] \forall p : \text{self.precondition} \mid p \neq \emptyset \bullet \\
\quad \forall mc : \text{self.mentalConstraint} \mid mc.\text{kind} \neq \emptyset \\
\quad \land mc.\text{kind} = \text{preCondition} \bullet \\
\quad p.\text{specification} = mc.\text{kind}.\text{specification}
\]

\[2\] \forall p : \text{self.postcondition} \mid p \neq \emptyset \bullet \\
\quad \forall mc : \text{self.mentalConstraint} \mid mc.\text{kind} \neq \emptyset \\
\quad \land mc.\text{kind} = \text{postCondition} \bullet \\
\quad p.\text{specification} = mc.\text{kind}.\text{specification}
\]

\[3\] \forall c : \text{context} \mid c \neq \emptyset \bullet \\
\quad \text{isKindOf}(c, \text{MentalSemiEntityType}) = \text{true}

Plan is a specialized class that inherits from Activity and ConstrainedMentalClass classes. Following invariants must be satisfied:

[1] Specification of the Constraint referred to by the precondition set is identical with the specification of the MentalConstraint of kind preCondition referred to by the mentalConstraint set, if the both are specified.

[2] Specification of the Constraint referred to by the postcondition set is identical with the specification of the MentalConstraint of kind postCondition referred to by the mentalConstraint set, if the both are specified.
10.4. Plans

[3] If the context for Plan is specified, it must be a MentalSemiEntityType.

10.4.2 CommitGoalAction

CommitGoalAction is a specialized CreateObjectAction (from UML) and AddStructuralFeatureValueAction (from UML), used to model the action of commitment to a Goal. 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 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).

For more details see [1, p. 284].

CreateRoleAction is introduced to model commitment actions within Activities (Plans).

CommitGoalAction

\[\ldots, \text{mentalProperty}, \text{mentalSemiEntity}, \text{goalInstance}, \text{goalType}\]

AddStructuralFeatureValueAction

CreateObjectAction

| mentalProperty : \(\mathbb{P}\) MentalProperty |
| mentalSemiEntity : \(\mathbb{P}\) InputPin \(\diamond\) |
| goalInstance : \(\mathbb{P}\) OutputPin \(\odot\) |
| goalType : \(\mathbb{P}\) Goal |

\#mentalProperty = 1
\#mentalSemiEntity = 1
\#goalInstance = 1
\#goalType = 1

[1] mentalSemiEntity.type \(\neq \emptyset\) \(\Rightarrow\) isKindOf(mentalSemiEntity.type, MentalSemiEntityType) = true

[2] goalInstance.type \(\neq \emptyset\) \(\Rightarrow\) conformsTo(goalInstance.type, goalType) = true

[3] mentalProperty.type \(\neq \emptyset\) \(\Rightarrow\) conformsTo(goalType, mentalProperty.type) = true

[4] \(\forall\) hc : self.activity().hostClassifier() \(\bullet\) isKindOf(hc, MentalSemiEntityType) = true

CommitGoalAction class inherits from AddStructuralFeatureValueAction and CreateObjectAction class. Following invariants must be satisfied:
10.5. Mental Relationships

[1] If the type of the InputPin referred to by the mentalSemiEntity set is specified, it must be a MentalSemiEntityType.

[2] If the type of the OutputPin referred to by the goalInstance set is specified, it must conform to the Goal referred to by the goalType set.

[3] If the type of the MentalProperty referred to by the mentalProperty set is specified, the Goal referred to by the goalType set must conform to it.

[4] CommitGoalAction can be performed only by a mental semi-entity.

10.4.3 CancelGoalAction

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 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). For more details see [1, p. 287]. CancelGoalAction is introduced to model de-commitment to goals.

\[
\text{CancelGoalAction} \\
\left\langle \ldots, \text{goalInstance} \right\rangle \\
\text{DestroyObjectAction}
\]

\[
\begin{align*}
\text{goalInstance} & : \mathbb{F} \text{ InputPin} \\
\#\text{goalInstance} & \geq 1 \\
[1] \forall \ gi : \text{goalInstance} & \ | \ gi.\text{type} \neq \emptyset \bullet \\
& \quad \text{isKindOf}(gi.\text{type}, \text{Goal}) = \text{true} \\
[2] \forall \ hc : \text{self.\text{activity}}().\text{hostClassifier}() & \bullet \\
& \quad \text{isKindOf}(hc, \text{MentalSemiEntityType}) = \text{true}
\end{align*}
\]

CancelGoalAction class inherits from DestroyObjectAction class. Following invariants must be satisfied:

[1] If the types of the InputPins referred to by the goalInstance set are specified, they must be Goals.

[2] CancelGoalAction can be performed only by a mental semi-entity.

10.5 Mental Relationships

The Mental Relationships package defines metaclasses used to model relationships between MentalStates which can support reasoning processes.
10.5.1 Contribution

Contribution is a specialized MentalRelationship and DirectedRelationship (from UML) used to model logical relationships between MentalStates and their MentalConstraints. The manner in which the contributor of the Contribution relationship (i.e. a MentalState referred to by the contributor 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 metaattribute 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 10.1.5. If contributor and/or beneficiary is a Belief, the contributorConstraintKind and/or the beneficiaryConstraintKind meta-attribute is unspecified. 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. For more details see [1, p. 289].

Contribution is introduced to model logical relationships between MentalStates and their MentalConstraints.
10.5. Mental Relationships

Contribution

\[(\ldots, \text{kind, contributorConstraintKind}, \text{beneficiaryConstraintKind, degree}, \text{beneficiary}, \text{contributor})\]

MentalRelationship

DirectedRelationship

kind : ContributionKind
contributorConstraintKind : seq MentalConstraintKind
beneficiaryConstraintKind : seq MentalConstraintKind
degree : seq ValueSpecification
beneficiary : P MentalState
contributor : P MentalState

\[
\#\text{contributorConstraintKind} \leq 1
\#\text{beneficiaryConstraintKind} \leq 1
\#\text{degree} \leq 1
\#\text{beneficiary} = 1
\#\text{contributor} = 1
\]

[1] isKindOf(\text{contributor, Belief}) = true
\lor isKindOf(\text{contributor, Contributor}) = true
\Rightarrow \text{contributorConstraintKind} \neq \emptyset

[2] isKindOf(\text{beneficiary, Belief}) = true
\lor isKindOf(\text{beneficiary, Contribution}) = true
\Rightarrow \text{beneficiaryConstraintKind} \neq \emptyset

Contribution class inherits from MentalRelationship and DirectedRelationship classes. Invariant [1] says that if the MentalState referred to by the contributor set is a Belief or a Contribution, the contributorConstraintKind set is unspecified. Invariant [2] formalizes the fact that if the MentalState referred to by the beneficiary set is a Belief or a Contribution, the beneficiaryConstraintKind set is unspecified.

10.5.2 ContributionKind

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. For more details see [1, p. 298].

ContributionKind is introduced to define possible kinds of Contributions.

\[\text{ContributionKind} ::= \text{sufficient} | \text{necessary} | \text{iff}\]

In Object-Z, ContributionKind is an enumeration, which has sufficient, necessary, and iff values.
Chapter 11

Ontologies

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

11.1 Basic Ontologies

The Basic Ontologies package defines the generic means for modeling of ontologies in AML, namely, ontology classes and their instances, relationships, constraints, and ontology utilities. Ontology models are structured by means of the ontology packages.

11.1.1 Ontology

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. For more details see [1, p. 300].

Ontology is introduced to specify a single ontology.

```
Ontology
```

```
Package
```

Ontology class inherits from Package class.

11.1.2 OntologyClass

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, for example, name, multiplicity, list of parameters, return value, or standard tagged values), the user is
allowed to extend this metainformation by adding specific tagged values. *OntologyClass* can also be used as an *AssociationClass*. For more details see [1, p. 300].

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

```
OntologyClass
   Class
```

*OntologyClass* inherits from *Class* class.

### 11.1.3 OntologyUtility

*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*. 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. For more details see [1, p. 302].

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

```
OntologyUtility
   Class
```

*OntologyUtility* class inherits from *Class* class.
Chapter 12

Model Management

The Model Management package defines the generic-purpose modeling constructs which can be used to structure AML models and thus manage their complexity and understandability.

12.1 Contexts

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.

12.1.1 Context

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. For more details see [1, p. 304].

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.

\[
\text{Context} = \{\text{situationConstraint}, \text{situationState}\} \\
\text{Package}
\]

\[
\begin{align*}
\text{situationConstraint} & : \mathbb{P} \text{ Constraint} \\
\text{situationState} & : \mathbb{P} \text{ State}
\end{align*}
\]

\[
\#\text{situationConstraint} \leq 1 \\
\#\text{situationState} \leq 1
\]

\[ [1] \ (\text{situationState} \neq \emptyset \land \text{situationContext} = \emptyset) \lor \ (\text{situationState} = \emptyset \land \text{situationContext} \neq \emptyset) \]

Context class inherits from Package class. Invariant [1] says that either the situationState or the situationConstraint can be specified.
Chapter 13

UML Extension for AML

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.

13.1 Extended Actor

Actor, being a specialized AutonomousEntityType, can:

- own MentalProperties,
- have Capabilities,
- be decomposed into BehaviorFragments,
- provide and/or use services (see section 9.4),
- observe and/or effect its environment (see section 9.5),
- play entity roles (see section 8.5),
- participate in social relationships (see section 8.5), and
- specify values of the meta-attributes defined by the SocializedSemiEntityType.

For more details see [1, p. 307].

Extension of UML Actor is introduced to allow the modeling of Actors as AutonomousEntityTypes.

\[
\begin{array}{c}
\text{Actor} \\
\text{AutonomousEntityType}
\end{array}
\]

Actor class inherits from AutonomousEntityType class.
13.2 Extended BehavioralFeature

BehavioralFeature, being a specialized Capability, can in addition to UML BehavioralFeature also specify meta-associations: inputs, outputs, pre-conditions, and post-conditions. For more details see [1, p. 308].

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.

\[
\text{BehavioralFeature} \subseteq \text{Capability} \\
\text{BehavioralFeature} = \emptyset
\]

BehavioralFeature is an abstract class, which inherits from Capability class.

13.3 Extended Behavior

Behavior, being a specialized Capability, can in addition to UML Behavior also specify meta-associations: inputs, outputs, pre-conditions, and post-conditions. For more details see [1, p. 309].

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.

\[
\text{Behavior} \subseteq \text{Capability} \\
\text{Behavior} = \emptyset
\]

Behavior is an abstract class that inherits from Capability class.
Part III

Abstract Multi-Agent Framework
Chapter 14

Concepts of AML

In order to properly understand AML, it is necessary to understand its underlying concepts. This chapter provides a description of the fundamental concepts used to describe an abstract metamodel of a MAS. The intention is not to provide a comprehensive metamodel for all aspects and details of a MAS (such as detailed architectural design, system dynamics, or operational semantics), but rather to explain the concepts that were used as the underlying principles of AML and influenced the design of comprised modeling constructs. [1]

Every section describes a part of the abstract MAS specified in OZ. Each component is specified independently in a separate subsection in following matter:

- A short informal definition of presented component.
- An Object-Z class schema that formalizes the described component.
- A natural language explanation of presented schema.

This chapter is related to chapter 5 Concepts of AML [1, p. 37] and to chapter 6 AML Modeling Mechanisms [1, p. 53].

14.1 Operation Templates

Following generic schemas are used as templates and provides basic maintain operations on sets.

```
AddElement [X, Y] ________________________________
\Delta(X)
y? : Y
\hline
y? \notin X
X' = X \cup \{y?\}
```

>AddElement is a generic Z schema that forms a template of adding an element of type Y into set expressed by X. Generic parameters X and Y are substituted by real values.
14.2 Multi-Agent System

This part of the conceptual MAS metamodel specifies the overall model of a multi-agent system.

14.2.1 MAS

MAS (multi-agent system) is a system composed of several agents, capable of mutual interaction. In the AML framework, a multi-agent system is an object that consists of, in addition to agents, other entity types, e.g. Environments or Resources (see section 14.3). In general we say that a multi-agent system comprises Entities. Physically, such a system can be deployed on several agent execution environments. This ensures the fact that MAS is specialized Object.

\[
\text{MAS} \overset{\text{MAS}}{\cong} \text{Init, AddEntity, RemoveEntity, AlterEntity}
\]

\[
\text{Object} \overset{\text{Object}}{\cong} \text{comprise} : \mathbb{P} \downarrow \text{Entity}
\]

\[
\text{Init} \overset{\text{Init}}{=} \emptyset
\]

\[
\text{AddEntity} \overset{\text{AddEntity}}{=} \text{AddElement}[\text{comprise}, \downarrow \text{Entity}] \land [\text{self} \notin y?.\text{hostedBy} \Rightarrow y?.\text{AddHost}(\text{self})]
\]

\[
\text{RemoveEntity} \overset{\text{RemoveEntity}}{=} \text{RemoveElement}[\text{comprise}, \downarrow \text{Entity}] \land [\text{self} \in y?.\text{hostedBy} \Rightarrow y?.\text{RemoveHost}(\text{self})]
\]
The Object-Z definition of MAS includes visibility list, inherited class Object, comprise attribute, which is a set of Entity (and all its sub-classes) instances. We also define basic operations of addition, removal, and alteration on all attributes. Init state schema identified by Init keyword declares the initial state of MAS class schema.

14.3 MAS Semi-entities and Entities

This part of the AML conceptual model of MAS deals with the modeling of constituents of a multi-agent system. MAS may consist of a set of interconnected entities of different types, namely agents, resources and environments. They are represented by concrete classes in the MAS conceptual metamodel. Furthermore, these entities are categorized, according to their specific characteristics, into several categories expressed in the conceptual metamodel by abstract classes used as superclasses to the concrete ones. In order to maximize reuse and comprehensibility of the concepts, AML defines several auxiliary abstract metamodeling concepts called semi-entities. Semi-entity is a modeling concept that defines certain features specific to a particular aspect or aspects of entities, but does not itself represent an entity. All entities inherit their features from semi-entities. Because semi-entities are abstractions, the metaclasses representing semi-entities in the MAS conceptual metamodel are abstract, and therefore they cannot be instantiated at a system’s run time. [1]

14.3.1 StructuralSemiEntity

StructuralSemiEntity represents the capability of an entity to have properties, to be decomposed into other StructuralSemiEntities, and to be linked to other StructuralSemiEntity. Each StructuralSemiEntity has structural capability and can be structured, internally and externally. Internal structure of StructuralSemiEntity is given by values of owned properties and by nesting of StructuralSemiEntities. External structure of StructuralSemiEntities is specified by means of links among StructuralSemiEntities. Slot represents a set of key-value pairs that are used to specify properties of its owner. Values of all properties of StructuralSemiEntity determine its state. In order to model hierarchical structures, StructuralSemiEntity can be nested, i.e. one StructuralSemiEntity can contain other StructuralSemiEntities. StructuralSemiEntity can also be linked to other StructuralSemi-Entities. A link represents a semantic relationship of two or more StructuralSemiEntities that know each other and can communicate.
### 14.3. MAS Semi-entities and Entities

**StructuralSemiEntity**

\[
\begin{aligned}
\text{StructuralSemiEntity} &= (\text{slot}, \text{consist}, \text{link}, \text{Init}, \text{AddProperty}, \text{RemoveProperty}, \text{AlterProperty}, \\
&\quad \text{AddStructuralSemiEntity}, \text{RemoveStructuralSemiEntity}, \\
&\quad \text{AlterStructuralSemiEntity}, \text{AddLinkTo}, \text{RemoveLinkTo}, \text{AlterLinkTo})
\end{aligned}
\]

**slot**: \(\mathbb{P} \text{Name} \times \text{Value}\)

**consist**: \(\mathbb{P} \downarrow \text{StructuralSemiEntity} \circ\)

**link**: \(\mathbb{P} \downarrow \text{StructuralSemiEntity}\)

\[
\text{StructuralSemiEntity} = \emptyset \\
\text{consist} \subseteq \text{link}
\]

**INIT**

\[
\begin{aligned}
\text{slot} &= \emptyset \\
\text{consist} &= \emptyset \\
\text{link} &= \emptyset
\end{aligned}
\]

**AddProperty**

\[
\Delta(\text{slot}) \\
\quad (n?, v?): \text{Name} \times \text{Value} \\
\quad (n?, v?) \notin \text{slot} \\
\quad \text{slot}' = \text{slot} \cup \{(n?, v?)\}
\]

**RemoveProperty**

\[
\Delta(\text{slot}) \\
\quad (n?, v?): \text{Name} \times \text{Value} \\
\quad (n?, v?) \in \text{slot} \\
\quad \text{slot}' = \text{slot} \setminus \{(n?, v?)\}
\]

**AlterProperty**

\[
\text{AlterProperty} \triangleq [\text{old}? : \text{Name} \times \text{Value}; \text{new}? : \text{Name} \times \text{Value}] \\
\quad \land \text{RemoveProperty}([\text{old}?] \wedge \text{AddProperty}([\text{new}?])
\]

**AddStructuralSemiEntity**

\[
\text{AddStructuralSemiEntity} \triangleq \text{AddElement}[\text{consist}, \downarrow \text{StructuralSemiEntity}]
\]

**RemoveStructuralSemiEntity**

\[
\text{RemoveStructuralSemiEntity} \triangleq \text{RemoveElement}[\text{consist}, \downarrow \text{StructuralSemiEntity}]
\]

**AlterStructuralSemiEntity**

\[
\text{AlterStructuralSemiEntity} \triangleq \text{AlterElement}[\text{consist}, \downarrow \text{StructuralSemiEntity}]
\]

**AddLinkTo**

\[
\text{AddLinkTo} \triangleq \text{AddElement}[\text{link}, \downarrow \text{StructuralSemiEntity}]
\]

**RemoveLinkTo**

\[
\text{RemoveLinkTo} \triangleq \text{RemoveElement}[\text{link}, \downarrow \text{StructuralSemiEntity}]
\]

**AlterLinkTo**

\[
\text{AlterLinkTo} \triangleq \text{AlterElement}[\text{link}, \downarrow \text{StructuralSemiEntity}]
\]

The Object-Z definition of \textit{StructuralSemiEntity} includes visibility list, \textit{slot} attribute that represents the ability of having property, \textit{consist} attribute - a set of all \(\downarrow \text{StructuralSemiEntity}\) instances, where that set is contained \((\circ)\) and \textit{link} attribute of all \(\downarrow \text{StructuralSemiEntity}\) instances. We also define basic operations of addition, removal, and alteration on all attributes.

Following Object-Z given sets are introduced.
14.3. MAS Semi-entities and Entities

[Name]

_Name_ is a given set, from which the names of all classes, attributes, operations, operation parameters, associations and roles are drawn.

[Value]

_Value_ is a given set, from which all kinds of values are drawn. This incorporates basic types as numbers, characters, structural types, instances of classes defined in this chapter, etc.

14.3.2 DeployableSemiEntity

_DeployableSemiEntity_ represents the capability of an entity to be deployed on one or more _AgentExecutionEnvironments_ (see section 14.5).

\[
\text{DeployableSemiEntity} \triangleq \text{AddElement}[\text{hosting}, \text{Hosting}] \\
\land \forall h : \text{hosting} \implies h.\text{deployableSemiEntity} = \text{self} \\
\text{INIT} \\
\text{hosting} = \emptyset \\
\text{AddHosting} \triangleq \text{RemoveElement}[\text{hosting}, \text{Hosting}] \\
\land \forall y : \text{deployableSemiEntity} = \text{self} : \\
\{ \text{host} : \text{Hosting} \mid \text{host} \in y?.\text{agentExecutionEnvironment}.\text{hosting} \land \\
\text{host} = y? \} = \emptyset \\
\Rightarrow y?.\text{agentExecutionEnvironment}.\text{AddHosting}(y?) \\
\text{RemoveHosting} \triangleq \text{RemoveElement}[\text{hosting}, \text{Hosting}] \\
\land \forall y : \text{deployableSemiEntity} = \text{self} : \\
\{ \text{host} : \text{Hosting} \mid \text{host} \in y?.\text{agentExecutionEnvironment}.\text{hosting} \land \\
\text{host} = y? \} \neq \emptyset \\
\Rightarrow y?.\text{agentExecutionEnvironment}.\text{RemoveHosting}(y?) \\
\text{AlterHosting} \triangleq [\text{old}? : \text{Hosting}; \text{new}? : \text{Hosting}] \\
\land \text{RemoveHosting}(\text{old}?) \land \text{AddHosting}(\text{new}?)
\]

_DeployableSemiEntity_ class schema includes visibility list, _hosting_ attribute - a set of _Hosting_ instances. The attribute _hosting_ in the _DeployableSemiEntity_ class corresponds to an attribute _deployableSemiEntity_ in the _Hosting_ class, indicating a bi-directional relationship between a _DeployableSemiEntity_ and _Hosting_. We also define basic operations of addition, removal, and alteration on all attributes.

14.3.3 CapableSemiEntity

_CapableSemiEntity_ represents the capability of an entity to possess capabilities.
14.3. MAS Semi-entities and Entities

**CapableSemiEntity**

\[ (\text{hasCapability}, \text{Init}, \text{AddCapability}, \text{RemoveCapability}, \text{AlterCapability}) \]

- **hasCapability**: \( \mathbb{P} \downarrow \text{Capability} \)
- CapableSemiEntity = \( \varnothing \)

**INIT**

- **hasCapability** = \( \varnothing \)

AddCapability \( \triangleq \) AddElement[hasCapability, \( \downarrow \) Capability]

RemoveCapability \( \triangleq \) RemoveElement[hasCapability, \( \downarrow \) Capability]

AlterCapability \( \triangleq \) AlterElement[hasCapability, \( \downarrow \) Capability]

**CapableSemiEntity** class includes visibility list, **hasCapability** attribute – set of all instances of class **Capability** or all **Capability**’s subclasses. We say that **CapabilitySemiEntity** has a set of abilities that can be performed. At last, we define basic operations of addition, removal, and alteration on **hasCapability** attribute.

We introduce given set of all constraints that can represent a **precondition** or **postcondition** in the next class schema.

**[Constraint]**

**Constraint** represents a given set of all constraints.

**Capability** is used to model an abstract specification of a behavior that allows reasoning about and operations on that specification. Technically, a capability represents a unification of common specification properties of UML’s behavioral features and behaviors expressed in terms of inputs outputs, pre- and post-conditions. [1]

**Capability**

\[ (\text{precondition}, \text{postcondition}, \text{Init}, \text{Run}, \text{input}, \text{output}, \text{evalConstraint}, \text{evalPrecondition}, \text{compute}, \text{determinePostcondition}) \]

- **precondition**: \( \mathbb{P} \) Constraint
- **postcondition**: \( \mathbb{P} \) Constraint
- **input**: seq Name \( \times \) Value
- **output**: seq Name \( \times \) Value
- **evalConstraint**: Constraint \( \rightarrow \) Boolean
- **evalPrecondition**: \( \mathbb{P} \) Constraint \( \rightarrow \) Boolean
- **compute**: seq Name \( \times \) Value \( \rightarrow \) seq Name \( \times \) Value
- **determinePostcondition**: (seq Name \( \times \) Value) \( \times \) \( \mathbb{P} \) Constraint \( \rightarrow \) \( \mathbb{P} \) Constraint

run \( \text{evalPrecondition} = \bigvee \text{evalConstraint}(c : \text{dom evalPrecondition}) \)
### 14.3. MAS Semi-entities and Entities

#### INIT

- \( \text{precondition} = \emptyset \)
- \( \text{postcondition} = \emptyset \)
- \( \text{input} = \emptyset \)
- \( \text{output} = \emptyset \)

#### Run

\[ \Delta(\text{postcondition}, \text{output}) \]

- \( \text{evalPrecondition}(\text{precondition}) = \text{true} \)
- \( \text{input} \neq \langle \rangle \)
- \( \text{output}' = \text{compute}(\text{input}) \)
- \( \text{postcondition}' = \text{determinePostcondition}(\text{input}, \text{precondition}) \)

*Capability* is an Object-Z class that includes visibility list, a set of attributes and functions, and defines operation Run, which represents execution of capability. *Capability* defines preconditions and functions `evalConstraint` and `determinePostcondition` to evaluate them. All preconditions are logically OR-ed.

In Object-Z, *Capability* can also be understood as an operation, that can be run. When the evaluation of preconditions is true and the sequence of input values is not empty then a computation occurs, which outputs postconditions and output values.

#### 14.3.4 BehavioredSemiEntity

*BehavioredSemiEntity* represents the ability of an entity to own *Capabilities*, interact with other *BehavioredSemiEntities*, provide and use *Services*, to observe and effect their environment by means of *Perceptors* and *Effectors*, and to be decomposed into *Behavior-Fragments*. For details see section 14.7.

#### 14.3.5 SocializedSemiEntity

*SocializedSemiEntity* represents the capability of an entity to form societies and can participate in social relationships. See section 14.4 *Social Aspects* for details.

#### 14.3.6 MentalSemiEntity

*MentalSemiEntity* represents the capability of an entity to possess (or to be characterized in terms of) mental attitudes, e.g. which information it believes in, what are its objectives, needs, motivations, desires, what goal(s) it is committed to, when and how a particular goal is to be achieved, which plan to execute, etc. For details see section 14.8 *Mental Aspects*.

#### 14.3.7 Object

*Object* represents all objects, which can exist in the system. Each object is a specialized *StructuralSemiEntity*, which can own capabilities (*CapableSemiEntity*), and can be deployed in one or more *AgentExecutionEnvironments* (*DeployableSemiEntity*).
14.3. MAS Semi-entities and Entities

Object

[... deployedAt, Init, Deploy, Terminate, Replace]

StructuralSemiEntity
CapableSemiEntity
DeployableSemiEntity

deployedAt : \mathcal{P} \text{AgentExecutionEnvironment}

\forall d : \text{deployedAt} \cdot \text{self} \in d.\text{deployed}

\text{INIT}

deployedAt = \emptyset

\text{Deploy} \triangleq \text{AddElement}[\text{deployedAt}, \text{AgentExecutionEnvironment}] \\
\quad \land [\text{self} \notin y?.\text{deployed} \Rightarrow y?.\text{DeployObject}(\text{self})]

\text{Terminate} \triangleq \text{RemoveElement}[\text{deployedAt}, \text{AgentExecutionEnvironment}] \\
\quad \land [\text{self} \in y?.\text{deployed} \Rightarrow y?.\text{TerminateObject}(\text{self})]

\text{Replace} \triangleq [\text{old'?}, \text{new'?} : \text{AgentExecutionEnvironment}] \\
\quad \land \text{Terminate(\text{old}?)} \land \text{Deploy(\text{new}?)}

Object is an Object-Z class that comprises visibility list, inherited classes – StructuralSemiEntity, CapableSemiEntity, DeployableSemiEntity – a set of attributes and operations that are identical to addition, removal, and alteration, but have different names. Each Object can be deployed in an AgentExecutionEnvironment. This capability is represented by deployedAt attribute and following condition \( \forall d : \text{deployedAt} \cdot \text{self} \in d.\text{deployed} \).

14.3.8 Entity

Entity represents specialized Object, which can be hosted by an multi-agent system (MAS).

Entity

[... hostedBy, Init, AddHost, RemoveHost, AlterHost]

\text{Object}

hostedBy : \mathcal{P} \text{MAS}

\text{ENTITY} = \emptyset

\text{INIT}

hostedBy = \emptyset

\text{AddHost} \triangleq \text{AddElement}[\text{hostedBy}, \text{MAS}] \\
\quad \land [\text{self} \notin y?.\text{comprise} \Rightarrow y?.\text{AddEntity}(\text{self})]

\text{RemoveHost} \triangleq \text{RemoveElement}[\text{hostedBy}, \text{MAS}] \\
\quad \land [\text{self} \in y?.\text{comprise} \Rightarrow y?.\text{RemoveEntity}(\text{self})]

\text{AlterHost} \triangleq [\text{old'?} : \text{MAS}; \text{new'?} : \text{MAS}] \\
\quad \land \text{RemoveHost(\text{old}?)} \land \text{AddHost(\text{new}?)}
The Object-Z definition of Entity includes visibility list, inherited Object class, hostedBy attribute that represents a set of all MAS instances, which own Entity. We also define basic operations of addition, removal, and alteration on hostedBy attribute.

### 14.3.9 BehavioralEntity

*BehavioralEntity* is an abstract specialized entity which represents entities having the features of *BehavioredSemiEntities* (see section 14.7 Behaviors) and *SocializedSemiEntities* (see section 14.4 Social Aspects), and can play entity roles (see section 14.4 Social Aspects).

### 14.3.10 Resource

*Resource* is a concrete specialized *BehavioralEntity* used to represent a physical or an informational entity within the system, with which the main concern is its availability and usage (e.g. quantity, access rights, conditions of consumption).

```
Resource               
  [(...)             
  BehavioralEntity  

Resource = Object-Z class, that inherits from BehavioralEntity. Visibility list is completly inherited from the superclass – this is denoted by ↑(…). As we have mentioned in section 5.2.7, the visibility list of a class is not inherited by default.

### 14.3.11 AutonomousEntity

*AutonomousEntity* is an abstract specialized behavioral entity and *MentalSemiEntity* (see section 14.8 Mental Aspects), used to represent 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. *AutonomousEntity* can be characterized in terms of its mental attitudes (see section 14.8 Mental Aspects).

```
AutonomousEntity       
  [(...)               
  BehavioralEntity    
  MentalSemiEntity    

AutonomousEntity = ∅
```
14.3.12 Agent

Agent is a concrete specialized AutonomousEntity representing a self-contained entity that is capable of autonomous behavior within its environment. An agent is a special object having at least the following additional features:

- 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.

Other features such as mobility, adaptability, learning, etc., are optional in the AML framework.

\[
\text{Agent} \\
|(...) \\
\text{AutonomousEntity}
\]

Agent is an Object-Z class that inherits visibility list and all functionality from AutonomousEntity class.

14.3.13 Environment

Environment is a concrete specialized AutonomousEntity representing a logical or physical surroundings of comprised entities which provides conditions under which the entities exist and function. Environment defines a particular aspect or 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. One entity can appear in several environments at once and one environment can comprise several entities. Environments are not considered to be static. Their properties, structure, behavior, mental attitudes, participating entities and their features, etc. can change over time.

\[
\text{Environment} \\
|(...) \\
\text{AutonomousEntity}
\]

The Object-Z class Environment with its visibility list is inherited from AutonomousEntity.

14.4 Social Aspects

The social aspects define concepts used to model organization structure of entities, they social relationships and the possibility to play (social) roles.
14.4.1 SocialRelationshipKind

SocialRelationshipKind is introduced to define allowed values for SocialRelationship. Similar to UML, in Object-Z can be defined enumeration types.

\[
\text{SocialRelationshipKind} ::= \text{peer} \mid \text{superordinate} \mid \text{subordinate}
\]

SocialRelationshipKind is an enumeration type, which can have peer, superordinate, and subordinate values.

14.4.2 SocialRelationship

SocialRelationship is a particular type of connection existing between SocializedSemi-Entities related to or having to deal with each other. A SocialRelationship is characterized by its kind. AML predefines two rather high abstract kinds, peer-to-peer and superordinate-to-subordinate. The set of supported SocialRelationshipKinds can be extended as required, e.g. by producer-consumer, competitors, or kinds of interpersonal relationships inspired by sociology, for instance intimate relationships, sexual relationships, friendship, acquaintanceship, brotherhood, etc.

\[
\text{SocialRelationship} \downarrow (\text{kind, socializedSemiEntity})
\]

\begin{align*}
\text{kind} & \colon \text{SocialRelationshipKind} \\
\text{socializedSemiEntity} & \colon \downarrow \text{SocializedSemiEntity} \\
\text{self} & \in \text{socializedSemiEntity}.\text{socialRelationship}
\end{align*}

\[
\text{INIT} \\
\text{kind.Init} \\
\text{socializedSemiEntity.Init}
\]

SocialRelationship is an Object-Z class representing an association class from UML. It comprises of visibility list, attributes and initial state schema.

14.4.3 SocializedSemiEntity

SocializedSemiEntity is a semi-entity used to represent the capability of an entity to form societies and to participate in social relationships with other socialized semi-entities.
14.4 Social Aspects

SocialAspects

SocializedSemiEntity

<table>
<thead>
<tr>
<th>(socialRelationship)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object</td>
</tr>
</tbody>
</table>

socialRelationship : P SocialRelationship

SocializedSemiEntity = ∅

INIT

socialRelationship = ∅

AddSocialRelationship ≜ AddElement[socialRelationship, SocialRelationship]
RemoveSocialRelationship ≜ RemoveElement[socialRelationship, SocialRelationship]
AlterSocialRelationship ≜ AlterElement[socialRelationship, SocialRelationship]

SocializedSemiEntity is an abstract Object-Z class that inherits from Object. It includes socialRelationship attribute, which is a set of SocialRelationship instances. This attribute defines a social relationship to other SocializedSemiEntities. We also define basic operations of addition, removal, and alteration on socialRelationship attribute.

14.4.4 BehavioralEntity

BehavioralEntity is an abstract specialized entity which represents entities having the features of BehavioredSemiEntities (see section 14.7 Behaviors) and SocializedSemiEntities (see section 14.4 Social Aspects), and can play entity roles (see section 14.4 Social Aspects).

BehavioralEntity

<table>
<thead>
<tr>
<th>(. . . , play)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
</tr>
<tr>
<td>BehavioredSemiEntity</td>
</tr>
<tr>
<td>SocializedSemiEntity</td>
</tr>
</tbody>
</table>

play : P EntityRole ⊗

BehavioralEntity = ∅

∀ p : play • p.player = self

INIT

play = ∅

AddPlay ≜ AddElement[play, EntityRole]

∧ [y?: player = self]

RemovePlay ≜ RemoveElement[play, EntityRole]

∧ [y?: player = self]

AlterPlay ≜ [old?: EntityRole; new?: EntityRole]

∧ RemovePlay(old?) ∧ AddPlay(new?)
BehavioralEntity is an abstract Object-Z class, which inherits from Entity, BehavioralSemiEntity, and SocializedSemiEntity. It includes play attribute, that enables the possibility to play different EntityRoles. We also define basic operation of addition, removal, and alteration on play attribute.

### 14.4.5 EntityRole

EntityRole is a concrete specialized StructuralSemiEntity (see section 14.3 MAS Semientities and Entities), BehavioralSemiEntity (see section 14.7 Behaviors), MentalSemiEntity (see section 14.8 Mental Aspects), SocializedSemiEntity, and DeployableSemiEntity, used to represent either a usage of structural properties, execution of a behavior, participation in interactions, capability of deployment, or possession of a certain mental state by a BehavioralEntity in a particular context (e.g. interaction or social). We say that the BehavioralEntity, called entity role player (or simply player), plays a given EntityRole. One BehavioralEntity can play several EntityRoles at the same time and can dynamically change them. The EntityRole exists only while a BehavioralEntity plays it. EntityRole is an abstraction of features required from the BehavioralEntities which can play it. Each EntityRole should be realized by a specific implementation possessed by its player. Thus an EntityRole can be used as an indirect reference to BehavioralEntities, and as such can be utilized for the definition of reusable patterns (usually defined at the level of types).

```
EntityRole ↷ (... , player)
StructuralSemiEntity
BehavioralSemiEntity
MentalSemiEntity
SocializedSemiEntity
DeployableSemiEntity

[ player : \ BehavioralEntity
  self ∈ player.play ]
```

The Object-Z class EntityRole inherits from StructuralSemiEntity, BehavioralSemiEntity, MentalSemiEntity, SocializedSemiEntity, and DeployableSemiEntity. Each EntityRole has a player that plays it.

### 14.4.6 OrganizationUnit

OrganizationUnit is a concrete specialized Environment type (see section 14.3 MAS Semientities and Entities) used to represent a social environment or its part. OrganizationUnits are usually used to model different kinds of societies, e.g. groups, organizations, institutions, etc. From an external perspective, OrganizationUnits represent coherent AutonomousEntities which can have external structure, perform behavior, interact with their environment, offer services (see section 14.7 Behaviors), possess mental attitudes (see section 14.8 Mental Aspects), play roles, etc. Properties and behavior of OrganizationUnits are both:
14.5. MAS Deployment and Mobility

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

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

\[
\text{OrganizationUnit} \searrow (\ldots)\text{Environment}
\]

*OrganizationUnit* is represented as an Object-Z class, that inherits from *Environment*.

### 14.5 MAS Deployment and Mobility

The MAS deployment specifies a set of concepts that are used to define the execution architecture of a MAS in terms of deployment MAS entities to a physical execution environment. The execution environment is modeled by one or more, possibly interconnected and nested, agent execution environments. The placement and operation of entities at agent execution environments is specified by concept of hosting. The AML deployment model supports also mobility, i.e. movement or cloning of entities among different agent execution environments, that is modeled by dynamic reallocation of hostings. A moving entity changes its present hosting by a new one located at another agent execution environment. Cloned entity creates its copy (called clone) with a new hosting placed at the same or different agent execution environment. [1]

#### 14.5.1 HostingKind

*HostingKind* is introduced to define possible values of the *kind* attribute of the *Hosting* relationship.

\[
\text{HostingKind} ::= \text{resident} \mid \text{visitor}
\]

*HostingKind* is defined as enumeration type, which can have *resident*, and *visitor* values.

#### 14.5.2 Hosting

*Hosting* is a relationship between an *DeployableSemiEntity* and the *AgentExecutionEnvironment* where the *DeployableSemiEntity* runs. It can be characterized by the *HostingKind*, which is one of the following:

- *resident* - the *DeployableSemiEntity* is perpetually hosted by the *AgentExecutionEnvironment*, or
- *visitor* - the *DeployableSemiEntity* is temporarily hosted by the *AgentExecutionEnvironment*. 
Hosting is an Object-Z class that forms an association class between DeployableSemiEntity and AgentExecutionEnvironment.

14.5.3 AgentExecutionEnvironment

AgentExecutionEnvironment is a concrete specialized StructuralSemiEntity and BehavioralSemiEntity (see section 14.7 Behaviors), used to represent an execution environment of multi-agent system. AgentExecutionEnvironment provides the physical infrastructure in which MAS DeployableSemiEntities can run. One DeployableSemiEntity can run in at most one AgentExecutionEnvironment at one time. It can run at one computational resource (computer) or is distributed among several nodes possibly connected by a network. It can provide (use) a set of services that DeployableSemiEntities use (provide) at run time. Owned hostings specify DeployableSemiEntities hosted by (running at) the AgentExecutionEnvironment.
14.6 Communicative Interactions

Communicative interactions specify a set of concepts defining communication between objects, entities, etc.

14.6.1 Message

Message is a specification of the conveyance of information from one instance to another, with the expectation that activity will ensue. [10]
14.6. Communicative Interactions

Message is represented as an Object-Z class that knows its sender, receiver, and encapsulates MessagePayload.

MessagePayload is a given set of all possible message payloads, that can be send in a message.

14.6.2 MultiMessage

MultiMessage is a specialized Message, which can be send to multiple receivers. If the toItSelf attribute is set to true and the sender belongs to the group of receivers, then the MultiMessage is sent also to its sender.

Object-Z class MultiMessage inherits from Message and overrides Message’s condition. MultiMessage can now be send to multiple receivers.

14.6.3 DecoupledMessage

DecoupledMessage is a specialized MultiMessage which is used to model a specific kind of communication, particularly the asynchronous sending and receiving.

DecoupledMessage is a specialized MultiMessage class.
14.6.4  SendMessageCapability

Send\(\text{MessageCapability}\) represents the capability to send message.

\[
\begin{array}{c}
\text{SendMessageCapability} \\
\text{(\ldots) Capability} \\
\text{input : } \mathbb{P} \downarrow \text{Message}
\end{array}
\]

Send\(\text{MessageCapability}\) is a specialized \(\text{Capability}\) that specifies the input set to be a set of \(\text{Message}\) instances.

14.6.5  ReceiveMessageCapability

Receive\(\text{MessageCapability}\) represents the capability to receive message.

\[
\begin{array}{c}
\text{ReceiveMessageCapability} \\
\text{(\ldots) Capability} \\
\text{output : } \mathbb{P} \uparrow \text{Message}
\end{array}
\]

Receive\(\text{MessageCapability}\) is a specialized \(\text{Capability}\) that specifies the output set to be a set of \(\text{Message}\) instances.

14.7  Behaviors

This part of the conceptual MAS metamodel specifies the concepts used to model behavioral aspects of MAS entities, namely:

- behavior abstraction and decomposition,
- communicative interactions,
- services, and
- observations and effecting interactions.

14.7.1  Perceptor

Perceptor is a means to enable its owner, \(\text{BehavioredSemiEntity}\), to observe, i.e. perceive a state of and/or to receive a signal from its environment (surrounding objects, entities, etc.).
14.7. Behaviors

\[
\text{Perceptor} \\
\text{↾} (\ldots, \text{owner}, \text{perceivingAct}) \\
\text{Object}
\]

\[
\text{owner} : \text{BehavioredSemiEntity} \\
\text{perceivingAct} : \mathcal{P} \downarrow \text{PerceivingAct} \\
\text{self} \in \text{owner}.\text{hasPerceptor}
\]

Perceptor is specialized Object, that has capability to perceive its environment. Each Perceptor has defined its owner.

PerceivingAct is a Perceptor’s capability to perceive.

\[
\text{PerceivingAct} \\
\text{↾} (\ldots) \text{Capability}
\]

PerceivingAct is a specialized Capability.

Effector is a means to enable its owner, BehavioredSemiEntity, to bring about an effect on others, i.e. to directly manipulate with (or modify a state of) some other objects, entities, etc.

14.7.2 Effector

\[
\text{Effector} \\
\text{↾} (\ldots, \text{owner}, \text{effectingAct}) \\
\text{Object}
\]

\[
\text{owner} : \text{BehavioredSemiEntity} \\
\text{effectingAct} : \mathcal{P} \downarrow \text{EffectingAct} \\
\text{self} \in \text{owner}.\text{hasPerceptor}
\]

Effector is specialized Object, that has capability to effect its environment. Each Effector has defined its owner.

EffectingAct is a Effector’s capability to perceive.

\[
\text{EffectingAct} \\
\text{↾} (\ldots) \text{Capability}
\]

EffectingAct is a specialized Capability.
14.7.3 Service

_Service_ is a coherent block of functionality provided by _BehavioredSemiEntity_, called _service provider_, that can be accessed by other _BehavioredSemiEntity_, called _service clients_.

\[
\text{Service} \\
| (\ldots, \text{owner}, \text{provide}) \\
\text{Object} \\
\]

\[
\begin{align*}
\text{owner} & : \text{BehavioredSemiEntity} \\
\text{provide} & : \text{P} \downarrow \text{ServiceFunctionality} \\
\text{self} & \in \text{owner}.\text{providesService}
\end{align*}
\]

_Service_ is a specialized _Object_, that has capability to run some functionality. This fact express the _provide_ attribute. Each Service has its _owner_.

_ServiceFunctionality_ is _Service_’s capability to provide functionality.

\[
\text{ServiceFunctionality} \\
| (\ldots) \\
\text{Capability}
\]

_ServiceFunctionality_ is a specialized Capability.

14.7.4 BehavioredSemiEntity

_BehavioredSemiEntity_ is a semi-entity used to represent the ability of an entity to have communicative capabilities, interact with other _BehavioredSemiEntity_, provide and use _Services_, to percept and effect, and to be decomposed into _BehaviorFragments_.

BehavioredSemiEntity


\[
\begin{align*}
\text{hasPerceptor} &\colon \mathbb{P} \text{ Perceptor} \\
\text{hasEffector} &\colon \mathbb{P} \text{ Effector} \\
\text{comprise} &\colon \mathbb{P} \text{ BehaviorFragment} \\
\text{interact} &\colon \mathbb{P} \downarrow \text{BehavioredSemiEntity} \\
\text{receiveMessageCapability} &\colon \mathbb{P} \downarrow \text{ReceiveMessageCapability} \\
\text{sendMessageCapability} &\colon \mathbb{P} \downarrow \text{SendMessageCapability} \\
\text{providesService} &\colon \mathbb{P} \text{ Service} \\
\text{usesService} &\colon \mathbb{P} \text{ Service}
\end{align*}
\]

\[\text{BehavioredSemiEntity} = \emptyset\]

\[
\begin{align*}
\text{Init} \\
\text{hasPerceptor} = \emptyset \\
\text{hasEffector} = \emptyset \\
\text{comprise} = \emptyset \\
\text{interact} = \emptyset \\
\text{receiveMessageCapability} = \emptyset \\
\text{sendMessageCapability} = \emptyset \\
\text{providesService} = \emptyset \\
\text{usesService} = \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{AddPerceptor} &\triangleq \text{AddElement}[\text{hasPerceptor}, \text{Perceptor}] \\
\text{RemovePerceptor} &\triangleq \text{RemoveElement}[\text{hasPerceptor}, \text{Perceptor}] \\
\text{AlterPerceptor} &\triangleq \text{AlterElement}[\text{hasPerceptor}, \text{Perceptor}] \\
\text{AddEffector} &\triangleq \text{AddElement}[\text{hasEffector}, \text{Effector}] \\
\text{RemoveEffector} &\triangleq \text{RemoveElement}[\text{hasEffector}, \text{Effector}] \\
\text{AlterEffector} &\triangleq \text{AlterElement}[\text{hasEffector}, \text{Effector}] \\
\text{AddBehaviorFragment} &\triangleq \text{AddElement}[	ext{comprise}, \text{BehaviorFragment}] \\
\text{RemoveBehaviorFragment} &\triangleq \text{RemoveElement}[	ext{comprise}, \text{BehaviorFragment}] \\
\text{AlterBehaviorFragment} &\triangleq \text{AlterElement}[	ext{comprise}, \text{BehaviorFragment}] \\
\text{AddInteraction} &\triangleq \text{AddElement}[	ext{interact}, \downarrow \text{BehavioredSemiEntity}] \\
\text{RemoveInteraction} &\triangleq \text{RemoveElement}[	ext{interact}, \downarrow \text{BehavioredSemiEntity}] \\
\text{AlterInteraction} &\triangleq \text{AlterElement}[	ext{interact}, \downarrow \text{BehavioredSemiEntity}] \\
\text{AddOwnedService} &\triangleq \text{AddElement}[	ext{providesService}, \text{Service}] \\
\text{RemoveOwnedService} &\triangleq \text{RemoveElement}[	ext{providesService}, \text{Service}] \\
\text{AlterOwnedService} &\triangleq \text{AlterElement}[	ext{providesService}, \text{Service}] \\
\text{AddUsedService} &\triangleq \text{AddElement}[	ext{usesService}, \text{Service}]
\end{align*}
\]
14.8 Mental Aspects

Object-Z class BehavioredSemiEntity includes the capability to perceive an effect (attributes hasPerceptor and hasEffector), to be decomposed into BehaviorFragments (comprise attribute), to interact with other BehavioredSemiEntity (interact attribute), to have the capability of sending and receiving messages (attributes receiveMessageCapability and sendMessageCapability), to provide its own Services (providesService attribute), and to use somebody else’s Services (usesService attribute). We also define basic operation of addition, removal, and alteration on all defined attributes.

14.7.5 BehaviorFragment

BehaviorFragment is a concrete specialized BehavioredSemiEntity used to represent a coherent re-usable fragment of behavior. It is used to decompose a complex behavior into simpler and possibly concurrently executable fragments. BehaviorFragment can be shared by several BehavioredSemiEntities and behavior of BehavioredSemiEntity can be (possibly recursively) decomposed into several BehaviorFragments.

14.8 Mental Aspects

Autonomous entities can be characterized by their mental attitudes (such as beliefs, goals, and plans), which represent their informational, motivational and deliberative states. This part of the conceptual MAS metamodel deals with modeling these aspects.

14.8.1 ContributionKind

ContributionKind defines three kinds of contribution.

ContributionKind ::= sufficient | necessary | iff

ContributionKind is an enumeration, which has sufficient, necessary, and iff values.

14.8.2 Contribution

Contribution represents a logical relationship between MentalStates. Contribution specifies the manner in which the contributor (a MentalState which contributes) influences its beneficiary (a MentalState which is contributed to). Contribution refers to the conditions which characterize related MentalStates (e.g. pre- and post-conditions, invariants, etc.)
and specifies their logical relationship in terms of logical implication. AML thus defines three kinds of contribution: necessary, sufficient, and equivalent. Contribution’s degree can be used to specify the extent to which the contributor influences the beneficiary.

Object-Z class Contribution represent association class between MentalState, which owns Contribution, and MentalState that is linked to Contribution by Contribution’s MentalState attribute. Each relationship (between two MentalStates) has degree and kind. Degree is given set of values, which specifies the extent to which the contributor influences the beneficiary.

\[ \text{Degree} \]

Degree is a given set of values.

### 14.8.3 MentalAttitude

MentalAttitude is a relationship between a MentalSemiEntity and a MentalState representing that the MentalSemiEntity possesses the MentalState as its MentalAttitude, i.e. it believes a Belief, is committed to a Goal, or is intended to perform a Plan. MentalAttitude is characterized by the degree attribute, of which semantics varies with concrete type of the linked MentalState. It represents either the subjective reliability or confidence of the linked MentalSemiEntity in the information specified by Belief, relative importance of a Goal, or preference of a Plan.

\[ \text{MentalAttitude} \]

Degree is a given set of values.
Object-Z class MentalAttitude represents association class between MentalState and MentalSemiEntity.

### 14.8.4 MentalState

MentalState is an abstract concept serving as a common superclass to all the metaclasses which can be used to specify MentalAttitudes of MentalSemiEntities. MentalState can be related by Contributions. MentalState referred to by several MentalSemiEntities simultaneously represent their common MentalStates, e.g. common Beliefs or common Goals.

```
MentalState

| (contribution, mentalSemiEntity, Init, AddContribution,
  RemoveContribution, AlterContribution, AddMentalSemiEntity,
  RemoveMentalSemiEntity, AlterMentalSemiEntity) |

contribution : P Contribution
mentalSemiEntity : P MentalAttitude

MentalState = ∅
∀ m : mentalSemiEntity • m.mentalState = self

INIT

contribution = ∅
mentalSemiEntity = ∅

AddContribution ≡ AddElement[contribution, Contribution]
  ∧ [y?.mentalState ≠ self]
RemoveContribution ≡ RemoveElement[contribution, Contribution]
  ∧ RemoveContribution(old?) ∧ AddContribution(new?)
AddMentalSemiEntity ≡ AddElement[mentalSemiEntity, MentalAttitude]
  ∧ [y?.mentalState = self]
RemoveMentalSemiEntity ≡ RemoveElement[mentalSemiEntity, MentalAttitude]
  ∧ [y?.mentalState = self]
AlterMentalSemiEntity ≡ [old? : MentalAttitude; new? : MentalAttitude]
  ∧ RemoveMentalSemiEntity(old?) ∧ AddMentalSemiEntity(new?)
```

MentalState is an abstract Object-Z class that comprises of contribution and mentalSemiEntity attributes. We also define basic operation of addition, removal, and alteration on all defined attributes.

### 14.8.5 MentalSemiEntity

MentalSemiEntity is a semi-entity used to represent the capability of an entity to possess MentalAttitudes by connection the entity to MentalStates.
14.8. Mental Aspects

\[
\begin{align*}
\text{MentalSemiEntity} & \cup \text{ mentalState, Init, AddMentalState, RemoveMentalState, AlterMentalState } \\
\text{mentalState} & : \mathbb{P} \text{ MentalAttitude} \\
\forall ms : \text{ mentalState} \bullet ms.\text{mentalSemiEntity} = \text{self} \\
\text{INIT} & \\
\text{mentalState} & = \emptyset \\
\text{AddMentalState} & \triangleq \text{ AddElement}[\text{mentalState}, \text{MentalAttitude}] \\
& \wedge [y?.\text{mentalSemiEntity} = \text{self}] \\
\text{RemoveMentalState} & \triangleq \text{ RemoveElement}[\text{mentalState}, \text{MentalAttitude}] \\
& \wedge [y?.\text{mentalSemiEntity} = \text{self}] \\
\text{AlterMentalState} & \triangleq [\text{old?} : \text{MentalAttitude}; \text{new?} : \text{MentalAttitude}] \\
& \wedge \text{RemoveMentalState(old?) \wedge AddMentalState(new?)}
\end{align*}
\]

\text{MentalSemiEntity} is an abstract Object-Z class that includes \text{mentalState} attribute and defines operation of addition, removal, and alteration on this attribute.

14.8.6 Belief

\text{Belief} is a concrete specialized \text{MentalState} used to model information which \text{MentalSemiEntities} have (believe) about themselves or their environment, and which does not need to be objectively true.

\[
\begin{align*}
\text{Belief} & \cup (\ldots) \\
\text{MentalState}
\end{align*}
\]

\text{Belief} class is a specialized \text{MentalState} class.

14.8.7 Goals

\text{Goal} is an abstract specialized \text{MentalState} used to model conditions of states of affairs, the achievement or maintenance of which is controlled by an owning (committed) \text{MentalSemiEntity}. \text{Goals} are thus used to represent objectives, needs, motivations, desires, etc. of \text{MentalSemiEntities}.

\[
\begin{align*}
\text{Goal} & \cup (\ldots) \\
\text{MentalState}
\end{align*}
\]

\text{Goal} class is a specialized \text{MentalState} class.
**14.9. Autonomy Aspects**

*DecidableGoal* is a concrete specialized *Goal* used to model goals for which there are clear-cut criteria according to which the *MentalSemiEntity* controlling the *Goal* can decide whether the *Goal* has been achieved or not.

\[
DecidableGoal \\
| (\ldots) \\
| Goal
\]

*DecidableGoal* class is a specialized *Goal* class.

*UndecidableGoal* is a concrete specialized *Goal* used to model *Goals* for which there are no clear-cut criteria according to which the *MentalSemiEntity* controlling the *Goal* can decide whether the *Goal* has been achieved or not.

\[
UndecidableGoal \\
| (\ldots) \\
| Goal
\]

*UndecidableGoal* class is a specialized *Goal* class.

**14.8.8 Plan**

*Plan* is a concrete specialized *MentalState* used to represent an activity (expressed e.g. by a series of steps) that a *MentalSemiEntity* is intended to perform.

\[
Plan \\
| (\ldots) \\
| MentalState
\]

*Plan* class is a specialized *MentalState* class.

**14.9 Autonomy Aspects**

In this section we present an optional extension of *AutonomousEntity*, which is based on belief-desire-intention reasoning. Belief-Desire-Intention (BDI) model has come to be possibly the best known and best studied model of practical reasoning agents. For more details see [9].

Following classes do not belong to the conceptual AML metamodel presented in [1], but rather demostrate its extensibility capabilities.

**14.9.1 BDIAutonomousEntity**

*BDIAutonomousEntity* is an abstract specialized *AutonomousEntity* that incorporates BDI model.
BDIAutonomousEntity
\[ (\ldots, \text{beliefs}, \text{goals}, \text{plans}, \text{alterBeliefs}, \text{alterGoals}, \text{alterPlans}, \text{alterMentalState}, \text{chooseGoal}, \text{choosePlan}, \text{chooseEffector}, \text{Init}, \text{ExamineEnvironment}, \text{Act}, \text{Activity}) \]

AutonomousEntity

**beliefs**: \( \mathbb{P} \) Belief

**goals**: \( \mathbb{P} \downarrow \) Goal

**plans**: \( \mathbb{P} \) Plan

**alterBeliefs**: \( \mathbb{P} \downarrow \text{PerceivingAct} \times \mathbb{P} \) Belief \( \rightarrow \) \( \mathbb{P} \) Belief

**alterGoals**: \( \mathbb{P} \) Belief \( \times \) \( \mathbb{P} \downarrow \) Goal \( \rightarrow \) \( \mathbb{P} \downarrow \) Goal

**alterPlans**: \( \mathbb{P} \) Belief \( \times \) \( \mathbb{P} \downarrow \) Goal \( \times \) \( \mathbb{P} \) Plan \( \rightarrow \) \( \mathbb{P} \) Plan

**alterMentalState**: \( \mathbb{P} \) Beliefs \( \times \) \( \mathbb{P} \) MentalAttitude \( \rightarrow \) \( \mathbb{P} \) MentalAttitude

**chooseGoal**: \( \mathbb{P} \downarrow \) Goal \( \times \) \( \mathbb{P} \) MentalAttitude \( \rightarrow \) \( \downarrow \) Goal

**choosePlan**: \( \mathbb{P} \) Plan \( \times \) \( \downarrow \) Goal \( \rightarrow \) \( \text{Plan} \)

**action**: \( \mathbb{P} \downarrow \) ActionCapability

**chooseAction**: \( \mathbb{P} \downarrow \) ActionCapability \( \times \) Plan \( \rightarrow \) \( \downarrow \) ActionCapability

\[
\text{BDIAutonomousEntity} = \emptyset
\]

dom \( \text{first alterBeliefs} \subseteq \text{self}.\text{hasPerceptor}.\text{perceivingAct} \)

dom \( \text{second alterBeliefs} \subseteq \text{beliefs} \)

\[
\forall (x_1, x_2, x_3) \in \text{dom alterPlans} \cdot x_1 \subseteq \text{beliefs} \land x_2 \subseteq \text{goals} \land x_3 \subseteq \text{plans} \]

dom \( \text{first alterMentalState} \subseteq \text{beliefs} \land \text{dom second alterMentalState} \subseteq \text{self}.\text{mentalState} \)

dom \( \text{first chooseGoal} \subseteq \text{goals} \land \text{dom second chooseGoal} \subseteq \text{self}.\text{mentalState} \)

ran \( \text{chooseGoal} \in \text{goals} \)

ran \( \text{choosePlan} \in \text{plans} \)

ran \( \text{chooseAction} \in \text{action} \land \text{dom second chooseAction} \subseteq \text{plans} \)

ran \( \text{chooseAction} \in \text{action} \)

**INIT**

\[
\text{beliefs} = \emptyset
\]

\[
\text{goals} = \emptyset
\]

\[
\text{plans} = \emptyset
\]

**ExamineEnvironment**

\[
\Delta(\text{beliefs}, \text{goals}, \text{plans}, \text{self}.\text{mentalState})
\]

\[
\#\text{self}.\text{hasPerceptor} \geq 1 \Rightarrow \text{beliefs}' = \text{alterBeliefs}(\text{self}.\text{hasPerceptor}.\text{perceivingAct}, \text{beliefs})
\]

\[
\text{beliefs}' \neq \emptyset \Rightarrow \text{goals}' = \text{alterGoals}(\text{beliefs}', \text{goals})
\]

\[
(\text{beliefs}' \neq \emptyset \land \text{goals}' \neq \emptyset) \Rightarrow \text{plans}' = \text{alterPlans}(\text{beliefs}', \text{goals}', \text{plans})
\]

\[
\text{beliefs}' \neq \emptyset \Rightarrow \text{self}.\text{mentalState}' = \text{alterMentalState}(\text{beliefs}', \text{self}.\text{mentalState})
\]
14.9. Autonomy Aspects

\[
\Delta \left( \text{goals, plans} \right)
\]

\[
\begin{aligned}
goals \neq \emptyset \\
goal : \downarrow \text{Goal} \bullet goal = \text{chooseGoal} (\text{goals}, \text{self.mentalState}) \\
goals' = \text{goals} \setminus \{ \text{goal} \} \\
plan : \text{Plan} \bullet \text{plan} = \text{choosePlan} (\text{plans}, \text{goal}) \\
plans' = \text{plans} \setminus \{ \text{plan} \} \\
a : \downarrow \text{ActionCapability} \mid a = \text{chooseAction} (\text{actions}, \text{plan}) \bullet \\
a. \text{Run}
\end{aligned}
\]

\[
\square (\Diamond (\text{Activity occurs}))
\]

\textit{BDIAutonomousEntity} is a specialized \textit{AutonomousEntity} that incorporates a set of instances of \textit{Belief} (\textit{beliefs} attribute), of \textit{Goal} (\textit{goals} attribute), and of \textit{Plan} (\textit{plans} instances). \textit{BDIAutonomousEntity} owns capability to act. This is depicted by the set of \textit{ActionCapability} instances (\textit{action} attribute). We also define a set of functions. These are described in subsequent operations.

\textit{ExamineEnvironment} is an operation of examining \textit{BDIAutonomousEntity}’s environment, of altering \textit{BDIAutonomousEntity}’s beliefs, goals, plans, and mentalStates. When \textit{BDIAutonomousEntity} has at least one Perceptor, then it runs perceiving\textit{Act} capability and using \textit{alterBeliefs} function, it alters its \textit{beliefs} attribute. \textit{BDIAutonomousEntity} must believe in something to alter its \textit{goals}. The alteration of goals is processed by \textit{alterGoals} function. To change \textit{plans} the \textit{alterPlans} function is called. \textit{MentalStates} of \textit{BDIAutonomousEntity} are changed by call of \textit{alterMentalState} function.

\textit{Act} operation chooses \textit{goal} and subsequently \textit{plan} that will be run. It also chooses an \textit{action} from the set of all possible actions that \textit{BDIAutonomousEntity} is able to perform and process \textit{Run} of selected \textit{ActionCapability}.

\textit{Activity} operation comprises of \textit{ExamineEnvironment} and \textit{Act} operation, which are run sequentially. That means, that when \textit{Activity} operation is called, then firstly the \textit{ExamineEnvironment} operation is processed and secondly the \textit{Act} operation is runned.

The \( \square (\Diamond (\text{Activity occurs})) \) temporal invariant says that always (\( \square \)) the \textit{Activity} operation eventually (\( \Diamond \)) occurs.

Lastly, we present the \textit{ActionCapability} class.

\textit{ActionCapability} is an abstract capability representing all possible actions that \textit{BDI-AutonomousEntity} can perform.

\[
\begin{aligned}
\text{ActionCapability} \\
\left\lfloor (\ldots) \right. \\
\text{Capability}
\end{aligned}
\]

\textit{ActionCapability} is a specialized \textit{Capability}. 
Part IV

Summary of Achievements
Chapter 15

Conclusions and Future Works

“Formal methods for software development are becoming increasingly necessary as software becomes an important part of everyday life. To handle the complexities inherent in large-scale software systems these methods need to be combined with a sound development methodology which supports modularity and reusability. Object orientation, based on the concept that systems are composed of collections of interacting objects whose behaviors are specified by classes, is such a methodology.”
— Graeme Paul Smith
An Object-Oriented Approach to Formal Specification, 1992

This thesis has presented a formal specification of Agent Modeling Language using the formal specification language Object-Z. We showed a way how to transform models based on UML Infrastructure and we demonstrated that this approach generic enough. Therefore, metamodels that exist purely in OMG metamodelling framework can be transformed analogously to Object-Z formal specifications. We also outlined a way how to formally specify OCL constraints. In this part of our work we see possible extensions of the formal specification of AML. One such extension is to formally specify classes taken from UML 2.0 metamodel and the functions defined in section 5.3 on page 16. In general, we can say that the resulting formal metamodel of AML can serve as a basis for further formal verification and validation, model-based testing, and possible re-engineering.

Our thesis also presented an abstract multi-agent system based on concepts originated in [1] that can serve as a start line for future multi-agent theories operating on aspects. The intention was not to provide a comprehensive metamodel for all aspects and details of MAS, but rather to explain the concepts that were used as the underlying principles of AML. To define autonomous behavior, we used BDI logic, but the extensibility of our model ensures addition of others, more specific behavioral models. We also presented simple life cycle by defining operations of addition, removal and alteration on objects. This simple maintenance of properties can also be extended in a more specific way. Another important area of future could be a formal specification of real time interactions between entities in our abstract multi-agent system. For this purpose the Timed Communicating Object-Z (TCOZ) would be appropriate. Finally, we can say that our abstract MAS can serve as a base for further formal or informal models of multi-agent systems.
Bibliography


Abstrakt


Kľúčové slová: AML, Agent Modeling Language, MAS, Multi-Agent Systems, Object-Z, OZ, formal specification.