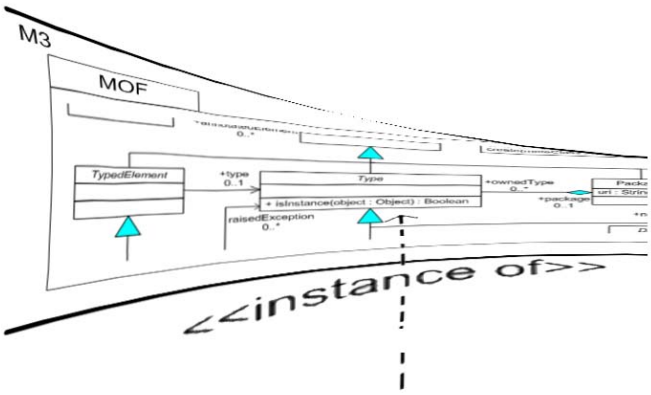
Meta Object Facility (MOF)

investigation of the state of the art



Ing. J.F. Overbeek
June 2006

Software Engineering,
Electrical Engineering, Mathematics
and Computer Science,
University of Twente

Dr. ir. A. Rensink
Drs. A.G. Kleppe
Prof. dr. ir. M. Akşit

Abstract

Model Driven Engineering (MDE) is the new trend in software engineering. MDE is the collection of all approaches that use models as core principle for software engineering. The Model Driven Architecture (MDA) is the proposed approach for MDE given by the Object Management Group (OMG). The core element of the MDA is the Model Object Facility (MOF), which is the object of study of this assignment.

In the history of software engineering, we are continuously searching for a technique that provides us with a better and more natural approach for defining a system in a more abstract way. The aim of the MDA is to reach an abstraction level that is more focused on defining the structure and behavior of the system, disregarding the underlying implementation technology. With the release of the Unified Modeling Language (UML), OMG has become the market leader in providing a modeling language for software engineering. Recently the OMG released the MDA, which covers the complete scope of using models in software engineering. Given the already earned market sector, the potential of the MDA can be significant. The MOF as a fundamental part in the MDA is therefore an important part to investigate.

The major goal of this thesis is codifying the usefulness and availability of the MOF. As approach for investigation, we will first investigate the scope of modeling in the domain of software engineering. Next, we investigate the standard itself, and after that the practical use of the standard.

The modeling approach is a good shift in software engineering to obtain a higher level of abstraction for defining a system. With the arrival of the MDA, the modeling concept for software engineering is standardized. Furthermore, the UML already booked success and is widely used. Based on this success the MDA has the potency to obtain the same position as UML. The same will hold for the MOF. Unfortunately, the MOF standard is not yet stable, and can still use some polishing especially in the area of the semantics.

Samenvatting

Model Driven Engineering (MDE) is de nieuwe tendens in softwaretechnologie. MDE is de verzamelnaam voor alle aanpakken die modellen als basis zien voor software technologie. De Model Driven Architecture (MDA) is de voorgestelde benadering voor MDE die door de Object Management Group (OMG) wordt gegeven. MDA bezit als kern de Model Object Facility (MOF), en deze is de doelstelling van deze opdracht.

In de geschiedenis van software technologie, zijn we onophoudelijk op zoek naar een techniek die ons een betere en natuurlijkere benadering geeft voor het bepalen van een systeem op een abstractere manier. Het doel van MDA is een abstractieniveau te bereiken dat zich meer concentreert op het bepalen van de structuur en het gedrag van het systeem los van de onderliggende implementatie technologie. Met de uitgave van de Unified Modeling Language (UML) is OMG marktleider geworden in het verstrekken van modelleringstalen voor softwaretechnologie. Onlangs heeft OMG de MDA uitgegeven, welke het volledige werkinggebied voor het gebruik van modellen in software technologie behandelt. Gezien het reeds verdiende marktsegment, kan de kracht van MDA significant zijn. MOF als fundamenteel deel in MDA is daardoor een belangrijk deel om te onderzoeken.

Het belangrijkste doel van deze thesis is het vastleggen van het nut en de bruikbaarheid van MOF. De aanpak van het onderzoek is als volgt. We zullen eerst het werkingsgebied van modellering op het gebied van software technologie onderzoeken. Vervolgens onderzoeken wij de standaard zelf, en afsluitend het praktisch gebruik van de standaard.

De modelleringbenadering is een goede verschuiving in software technologie om een hoger niveau van abstractie te verkrijgen voor het bepalen van een systeem. Door de komst van de MDA zijn de modelleringconcepten voor softwaretechnologie gestandaardiseerd. Verder boekte UML al reeds succes en wordt al veel gebruikt. Gebaseerd op dit succes heeft MDA de kracht om dezelfde positie te verdienen als

UML. Hetzelfde zal voor MOF gelden. De MOF standaard is alleen nog niet stabiel, en kan nog wat verbetering gebruiken vooral op het gebied van de semantiek.

Acknowledgements

I would like to thank all the people that made it possible to finish my master degree. First, I would like to thank all colleague students that helped me in finishing courses. Second, I would like to thank my supervisors Arend Rensink and Anneke Kleppe. They gave me lots of freedom, and the opportunity to create my own research. They made the lonely job of writing my thesis passable. Third, I would like to thank my family, my friends, and especially my girlfriend they gave me enough joy in live to get over difficult moments, which gave me the power to continue the job.

List of contents

1

Introduction

1.1 Background

Model Driven Engineering (MDE) is the new trend in software engineering. MDE is the collection of all approaches that use models as a core principle for software engineering. The Model Driven Architecture (MDA) is the proposed approach for the MDE given by the Object Management Group (OMG). The core element of the MDA is the Model Object Facility (MOF), which is the objective of this assignment.

Why modeling?
If we look to the history of software engineering, we can detect that we are continuously searching for a technique that provides a better and more natural approach for defining a system. For example, the introduction of the first FORTRAN compiler in 1957 brought a shift in defining a system in a more abstract way. At this point programmers were able to specify what a machine should do rather than how the machine should do it. (Atkinson and Kühne [38]) After the introduction of the first compiler the aim to develop a technique that provides the ability to define a system in an even higher abstraction level continues, for example with the arrival of procedural and object-oriented languages. The aim of the MDA is to reach an abstraction level that is more focused on defining the structure and behavior of the system disregarding the underlying implementation technology.

Why MOF?
With releasing the first Unified Modeling Language (UML) OMG became the market leader in providing a modeling language for software engineering. Until now UML is still the most used modeling language. Recently the OMG released the MDA, which covers the complete scope of using models in software engineering. Given the already earned market sector the potency of the MDA can be significant. The MOF as a fundamental part in the MDA is therefore an important part to investigate.

The current purpose of the MOF, given by the OMG [10], is to enable the development and interoperability of model and metadata driven systems, such as modeling and development tools, data warehouse systems and metadata repositories. For realizing this, MOF provides a metadata management framework, and a set of metadata services. The current released version of the MOF is 2.0, and originates from a previous version and is a response to the so-called Request For Proposals (RFPs).

1.2 Goals

The major goal of this thesis is codifying the usefulness and availability of the MOF. We can further specify this goal into three sub goals, namely basic concepts, MOF standard, practical use.

Basic concepts
The MOF is nowadays the fundamental part of the MDA. This is a relatively new technique in the software engineering, and especially as approach for defining a complete system. By means of the introduction of this technique, concepts as models, metamodels and modeling are getting a meaning according the scope of the software engineering. The goal is to investigate the meaning of these concepts according to software engineering.

MOF standard
The MOF standard is specified in the MOF specification. The goal is to investigate the most recently released MOF specification and analyze the structure and behavior of all concepts introduced in this specification.

Practical use
The MOF standard can be practically used by building implementations based on the standard. MOF compliant implementations can be checked on the interpretation and use of the MOF standard. The goal is to investigate these implementations on their interpretation and use of the standard.

1.3 Approach

For investigating the basic concepts we are mainly concentrated on the modeling technique for software engineering in general. The approach is to investigate the modeling technique in a broader sense, and not purely concentrated on the modeling technique as introduced by the OMG. This provides a description of the theoretical ideas behind modeling in contents of software engineering. The modeling approach uses models as first class concepts in the design of software. Therefore, we first define what a models is, and continue with describing the meaning of the model. In the current software engineering a programming language has a dominant roll in the design of software. Therefore, we describe what a language is, and continue with what its roll is according to modeling. In this research part, the meaning of metamodels, metamodeling and other related aspects is described as well. Together, this provides the theoretical background to investigate the MOF standard.

The idea for investigating the MOF standard is to get an exact view of how to use the standard. We analyze the most recent version of the MOF. However, for briefly describing how the MOF is originated, we investigate the preceding releases of the standard as well. For analyzing the MOF standard, we first concentrate on analyzing the architecture, after we continue with analyzing the formalism. The research on the architecture provides an overview of all concepts used, and their correspondent meaning. In analyzing the MOF formalism, the specification techniques used in the standard are described. This continues with a detailed description of essential concepts as introduced in the standard.

The next approach for investigation is more concentrated on using the MOF. In this investigation we are building metamodels based on the MOF. First, we analyze metamodels that are already defined based on the MOF. The most common used metamodels that are based on the MOF metamodel are UML and CWM, but the MOF itself as well. We choose to investigate the self-describing ability of the MOF in more detail. The MOF is a compact metamodel compared with the UML and CWM. Furthermore, investigating the self-describing ability of the MOF is more inline of the research done before. Second, we define a new metamodel based on the MOF. Besides describing how to define a metamodel, we also describe how to use the defined metamodel. Therefore, we instantiate some example models based on this metamodel, and further instantiate them as runtime models. With this we show the usage of the complete metamodel hierarchy.

The last approach for investigation is concentrated on related work in the area of the MOF. In this part we summarize the specification that are closely related to the MOF, tools that implemented the MOF standard, MOF metamodels, and related approaches. This provides an overview of the current state of the MOF standard. Furthermore, we can see in which area the MOF is already used.

1.4 Structure of the report

Chapter 2 describes the basic concepts for the MDE approach. This chapter will discuss the basic concepts needed for modeling. First, we will concentrate on a model, with the objective to explain what the meaning of a model is. Next, we will concentrate more on the aspects that are related to a model.

Chapter 3 describes the MOF standard. In this chapter, we will briefly describe the history of MOF. After that, the architecture used in the metamodel is considered, and the last section discusses the specification approach used to define the MOF 2.0.

Chapter 4 describes an example compliant with the MOF standard. In this chapter, a small subset of the electrical diagrams domain is defined as a metamodel based on the MOF. This chapter will further explain the instantiation of models based on this metamodel.

Chapter 5 describes related work in the area of the MOF. In this chapter, we will mention related work according to the MOF. As related work we can distinguish the following categories: MOF specifications series, compliant tools, MOF metamodels, and related approaches.

Chapter 6 describes the conclusion and recommendation made based on the investigation.

2

Basic Concepts

This chapter will discuss the basic concepts needed for modeling. First, we will concentrate on a model, with the objective to explain what the meaning of a model is. Next, we will concentrate more on the aspects that are related to a model.

2.1 What is a model?

The term model is applicable in a broad area, which leads to many definitions. For example, a definition of model according to Benyon is [25], *"A model is a representation of something, constructed and used for a particular purpose.*" The process of making a model is called *modeling*. A *modeler* constructs the model. The *interpreter* is using the model. Both modeler and interpreter have a particular *purpose* for the construction or use of the model. The model is always the representation of *something*.

A model on its own has no *meaning*. The meaning of the model is related to the situation and context wherein the model is used. Stachowiak [26] stated this as the pragmatic feature of the model. Like information and data [22], the data is the syntactic representation of information. Data on its own has no meaning, but in combination with an interpretation, the information behind it can be extracted and understood.

According to Benyon's definition of a model, we can detail the pragmatic use of the model as shown in Figure 1.

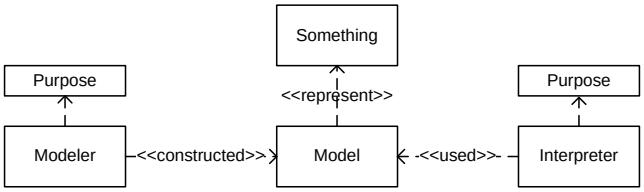


Figure 1 : Pragmatic use of the model

Let us illustrate this in more detail in the following example. A manufacturer of inverters would like to build a model of the inverter, to show how to work with the inverter. This is done for the customer that needs to know how to work with the inverter. The manufacturer decides to model the inverter as a truth table, which explains the working of the inverter. According to this, we have the following situation and context wherein the model is used:

- *Something* is the inverter.
- *Modeler* is the manufacturer.
- *Purpose* of the modeler according the model is, explaining the working of the inverter.
- *Interpreter* is the customer.
- *Purpose* of the interpreter according the model is, knowing the working of the inverter.
- *Model* is a truth table, showing the behavior of the inverter.

The Figure 2 shows the complete pragmatic use of the model.

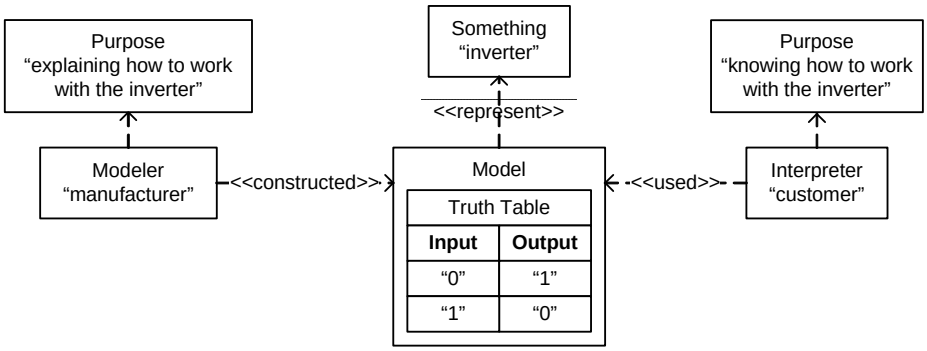


Figure 2 : Manufacturer of inverters example

2.2 Meaning of a model

In this section we will explain the meaning of a model in more detail. The meaning is related to the pragmatic users of the model. The modeler and interpreter will have an important role in the meaning of the model.

The modeler as constructor of the model will define together with constructing the model the meaning of the model. The modeler will construct the model in such way that based on the representation the meaning can be extracted. Therefore, the modeler is using already *commonly understood* concepts. As with the manufacturer of inverter, the manufacturer modeled the inverter in a commonly understood way for defining logical gates.

The role of the interpreter is to extract the meaning from the model. The interpreter is only capable of extracting the correct meaning if the interpreter has the same common understanding of the concepts used for the model.

The exchange of a model between a modeler and an interpreter is called *communication*. In the case of a modeler and an interpreter, the modeler is communicating with the interpreter. We can assign communication with a degree of *meaning*. The degree of meaning can be fuzzy, but we can at least define a minimum and maximum degree of meaning. The minimum degree of meaning is called *meaningless*, and maximum degree is called *meaningful*. If the modeler communicates with the interpreter, the modeler has a purpose for communicating. The communication between the modeler and interpreter is meaningful if the purpose is obtained, if the purpose is not obtained the communication is meaningless.

In the case of the manufacturer of inverters, if the customer knows how to work with the inverter based on the model, the communication between the manufacturer and the customer is meaningful. So, if we can check whether the customer actually knows how to work with the inverter, we can validate that the communication was meaningful.

The manufacturer predicts what the customer has as common understanding, and therefore what the interpreter can interpret. If the manufacturer can perfectly predict the common understanding of the customer, the chance for meaningful communication will increase. To enlarge the chance that the customer can interpret the model, the manufacturer can refer to a description of the notation of the model. This can be useful if the notation of a truth table is new for the customer. Therefore, the customer should be capable of interpreting the description of the truth table, otherwise we need again a description of the description of a truth table.

2.3 Language

In the section 2.2, we introduced the common understanding concept. In this section, we will discuss how to describe a common understanding.

For structurally describing something, we use a *language*. A language can be compared with the common understanding as described in section 2.2. The language is used for communication, and will at least need the following concepts. A language needs a concrete notation, which can be stored or transported. Furthermore, an interpretation is needed that will explain the meaning of the language constructs. These definitions are the fundamental concepts of a language, and are described as *syntax* and *semantics* [22]. The syntax of the language defines the notation, and the semantics describes the meaning of the notation.

Both syntax and semantics can be divided into aspects that are more specific. For the syntax those aspects are *concrete syntax*, *syntax mapping*, and *abstract syntax*, and for the semantics those aspects are *semantic mapping* and *semantic domain*. Those aspects are related to each other in some way. The Figure 3 shows an overview of those aspects and the relation with each other.

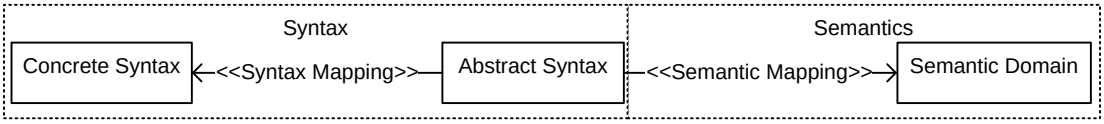


Figure 3 : Overview syntax and semantic specific aspects

As shown in Figure 3, the syntax of the language is divided into *concrete syntax* and *abstract syntax*. Where the concrete syntax defines the physical notation, the abstract syntax defines the structure of the notation. The structure of the notation is defined independently of the physical notation. Both syntaxes are mapped to each other by means of the *syntax mapping*, which provides the ability for defining a program using the physical notation according the abstract syntax.

For describing the meaning of the language the semantics are used, which describes the meaning in terms of the concepts that are already well-defined and well-understood. The well-defined and well-understood concepts are covered in the *semantic domain*, which is part of the semantics. For the semantic domain, we can use a variety of notations, like natural language or mathematical definitions. The abstract syntax is mapped to the semantic domain. This provides the abstract syntax with a well-defined and well-understood meaning.

As for everything we would like to describe, we need a language for describing it. In the case of the defined language aspects, it is not necessary that the same language is capable of describing each aspect.

The language used for describing models is called a *modeling language*. We can view the modeling language as a pragmatic user of the model, as shown in Figure 4 a.. The relationship between the modeling language and the model is that the model is expressed by using the modeling language. The modeler and interpreter need an understanding of the modeling language. The modeler can construct the model, based on this understanding. For the interpreter, the understanding will provide the ability to extract the correct meaning of the model. The relationship between the interpreter and modeling language is shown in Figure 4 b..

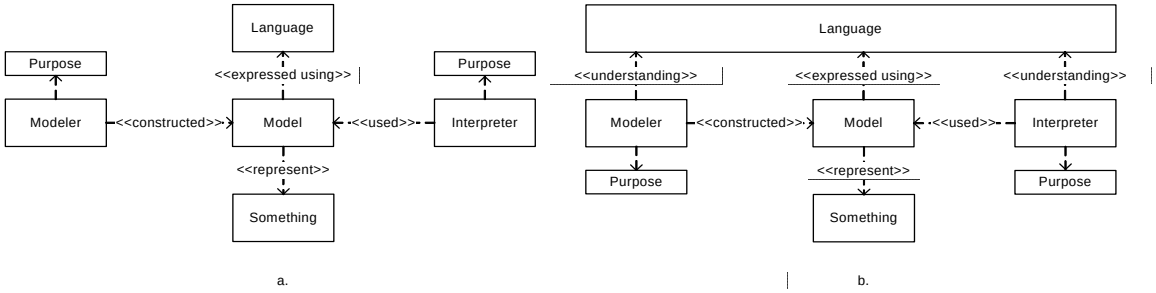


Figure 4 : Language as a pragmatic user of the model

2.4 Metamodel

A model that represents a modeling language is called a metamodel [24]. *Meta* is Greek for about or beyond, and is used for describing something. In the case of a metamodel it describes the possible models that can be expressed using the language, as shown in Figure 5. The model is an instantiation based on the metamodel. The relationship between a model and metamodel is called an *instance of* relationship.

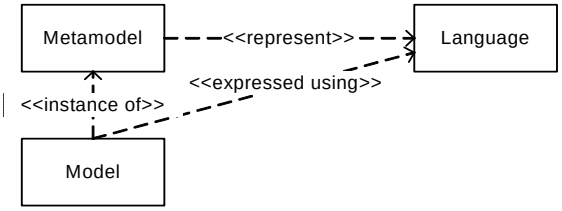


Figure 5 : The relationship between metamodel, model, and language

In section 2.3, we divided the language into aspects, as concrete syntax, syntax mapping, abstract syntax, semantic mapping, and semantic domain. The literature generally not agrees what should be part of the metamodel. In our opinion, the metamodel will at least represent the abstract syntax; the other language aspects are optional, but can be part of the metamodel, as shown in Figure 6.

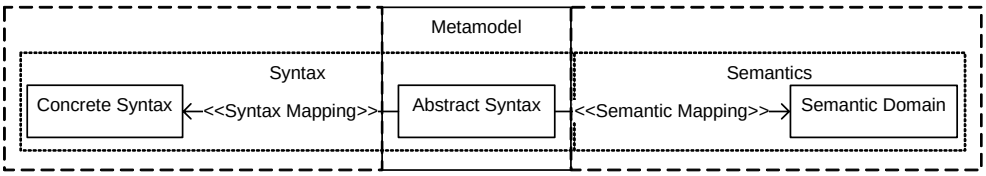


Figure 6 : Metamodel representing a language

Let us illustrate this in more detail with an example. In section 2.1 we modeled a representation for an inverter. We will now use this for building a metamodel that represents the modeling language for modeling inverters. The metamodel of the inverter contains anyhow the abstract syntax of the modeling language. The Figure 7 shows the abstract syntax for the inverter.

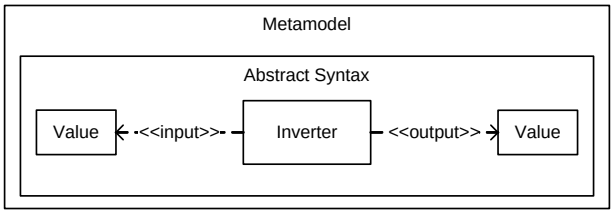


Figure 7 : Metamodel representing abstract syntax inverter modeling language

The truth table for the inverter, as defined in section 2.1, is the inverter explained in well-defined and well-understood concepts. According to this, we can use the truth table as semantic domain and map it with the

abstract syntax of the inverter, which will enlarge the metamodel with semantics. Figure 8 shows the semantic mapping between the abstract syntax and the semantic domain of the inverter.

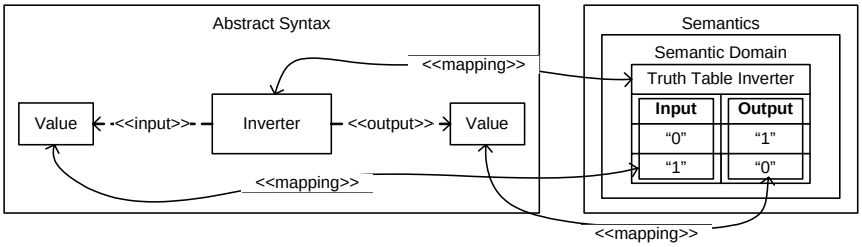


Figure 8 : Semantic mapping of the inverter modeling language

We can define the concrete syntax for the inverter as well. We can take, for example, the ANSI/IEEE notation for the inverter. By means of the syntax mapping, we can map the concrete syntax with the abstract syntax, as shown in Figure 9.

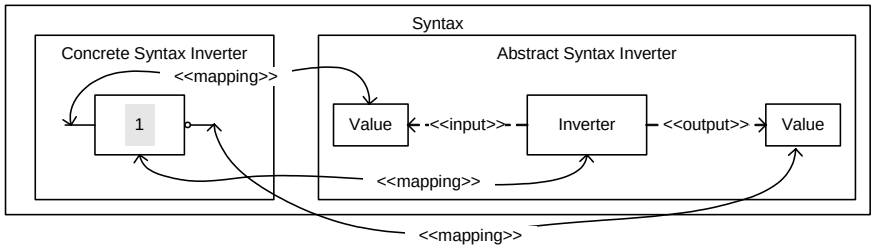


Figure 9 : Syntax mapping of the inverter modeling language

2.5 Runtime

The model is the representation of a system. The runtime is the representation an interpreter that interpreting the system. An important aspect in this is the *time*, the interpretation occurs always in time. The most basic form of time can be defined in three phases, namely *future*, *present*, and *past*. According to the interpreter, we have the following situations; the interpreter will interpret the system (future), the interpreter is interpreting the system (present), and the interpreter did interpret the system (past).

For example we model an inverter with input value "1". The inverter is an instantiated inverter based on the inverter metamodel as defined in section 2.4. The interpreter will use the model and interpret it according to the inverter metamodel. The modeled inverter is shown in Figure 10.

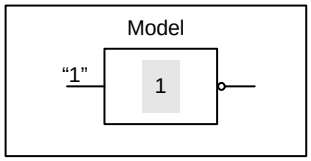


Figure 10 : Modeled inverter

The representation of the runtime is not defined in the metamodel. For viewing the runtime as a model, we use a combination of the defined abstract syntax and semantic domain. The inverter is replaced by the truth table, as defined in the semantic domain, for exactly showing which decision the inverter makes. According to the time definitions, we can make three models showing the interpretation of the modeled inverter, namely future inverting, present inverting, and past inverting. The complete runtime view of the interpretation of the modeled inverter is shown in Figure 11.

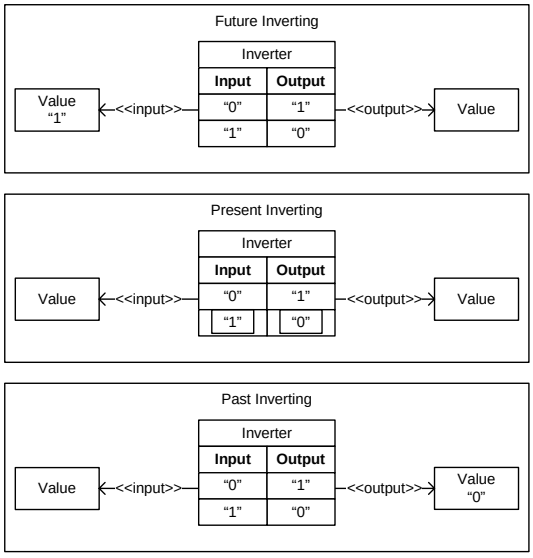


Figure 11 : Runtime view of the inverter

In this case the present form of the inverting has a very simple representation and can be shown in one figure. When we have a more complex model, we can have multiple views of the present process of interpreting the model.

2.6 Hierarchy

The involving parts for modeling are forming a *modeling hierarchy*. The hierarchy is divided into layers and each layer will contain a certain involving part. As described before we can have a metamodel for a model, which is located in the *meta-layer* of the modeling hierarchy. The model constructed by the modeler and interpreted through the interpreter is located in the *model-layer* of the modeling hierarchy. The view of decisions and actions that are made by the interpreter during modeling is located in the *runtime-layer*. In Figure 12 the complete modeling hierarchy is given.

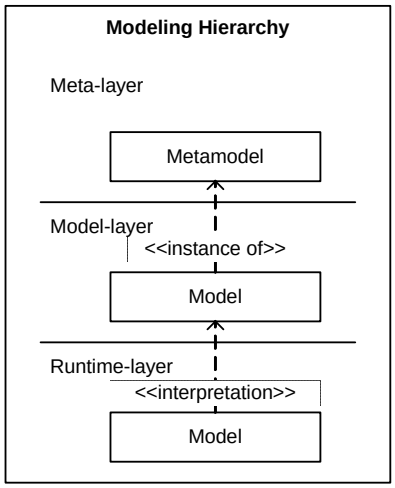


Figure 12 : Modeling hierarchy modeler

The number of meta-layers in a modeling hierarchy can be infinitive, which means that above a meta-layer always another meta-layer can appear. Normally we work with a fixed number of meta-layers.

If in a modeling hierarchy all models are described by a metamodel located in the meta-layer above, the modeling hierarchy is called *strict*. In a modeling hierarchy with infinity layers there is always a meta-layer

above a meta-layer, and can be strict if for each layer it is true that the containing models are described in the meta-layer above. In the case we have a fixed number of layers, the hierarchy always ends with a *top-layer*. Looking from the perspective of the modeling hierarchy, the top-layer will not have a meta-layer above it that describes it, and therefore the hierarchy cannot be strict.

If we take another perspective, we can define two ways of describing the top-layer of the metamodeling hierarchy. We will use the taxonomy as introduced by Gitzel and Hildenbrand [5]. The first option is taking another language that will describe the top-layer, but this language is not covered in the modeling hierarchy. We call this an *axiomatic-top-layer*. The hierarchy with an axiomatic-top-layer can never be strict. The second option is a top-layer capable of describing itself, which results in the fact that no additional language is needed. We call this a *recursive-top-layer*. (See section 2.8 for more details about recursive top layers) The hierarchy with a recursive-top-layer can be strict.

2.7 Meta Metamodel

A meta metamodel is a specialized metamodel that describes other metamodels. The position in the modeling hierarchy defines if a metamodel is a meta metamodel. The model in a meta-layer that is directly above model-layer is called a metamodel. The model in a meta-layer that has a meta-layer below it is called a meta metamodel.

We can of course continue this terminology by calling the model in the meta-layer above two meta-layers a meta meta metamodel. In this thesis, we will only speak about metamodels. The reason for this is that besides the position in the modeling hierarchy there are no differences between these models.

2.8 Reflection

In defining a language, reflection is one technique to provide an open system. Smith first introduced reflection in 1982 [2]. He defined reflection as:

"In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures".

A reflective system defines a representation of its own behavior and structure, which provides the ability to access, reason about and alter its own interpretation [3]. This representation, called self-representation, has a causal connection with the system. A causal connection will hold if the domain represented by the system and internal structure and behavior of the system is linked in such a way that when one form is changing, the other form is changing as well. So if the self-representation has a causal connection with the system, the self-representation will always be an accurate representation of the system.

The reflective ability of a system can be a *complete* or *partial* representation of its internals. We will call a system with complete representation abilities a *fully reflective system* and a system with partial representation abilities a *system with reflective facilities* [19]. The most commonly used reflective facilities are structural reflection and behavioral reflection, where structural reflection concerns the internal architecture of the system and the behavioral reflection concerns the internal actions in the system.

The main purpose of building a reflective system is self-inspection and self-adaptation [19]. The open internal structure and behavior provide the ability for inspecting what is happening in the system. This is useful in monitoring certain activities in the system, for example, in building a debugging tool where you would like to rely on the meta-objects for accessing the implementation details. The adaptation usually has the form of change or addition of new features to the system. This can be used, for example, in optimizing systems, like changing the quality of service of video on detecting quality of service violations.

The reflective technique is introduced in metamodeling as well. In the case of metamodeling, you would like to build the metamodel as a model with self-representation capability. Building the metamodel as self-

representing model will prove that the metamodel is expressive enough for defining itself. Therefore, no additional language is needed in building the metamodel, which means that the metamodel can be the highest layer in the metamodeling hierarchy. In which degree the metamodel is reflective depends on what the metamodel is representing. For example if the metamodel is concerned with defining an abstract syntax, the metamodel will have the structural reflective facility.

Because of the fact that the self-representation of the metamodel is built as a model based on the metamodel, the metamodel and the model are causally connected. This will provide the ability that when the metamodel is changing the model will change as well, and vise-versa.

For explaining reflection in more detail, we will consider the following examples. First, we will show the ability of building a self-representing model for a metamodel. Through structural reflection, we will define the self-representing model by only instantiating constructions defined in the metamodel. The example metamodel we will use, can define elements that can be connected with each other through relationships, as shown in Figure 13. For defining the self-representing model, we do not have to introduce any new constructions, instead we can instantiate each construction based on the defined ones in the metamodel.

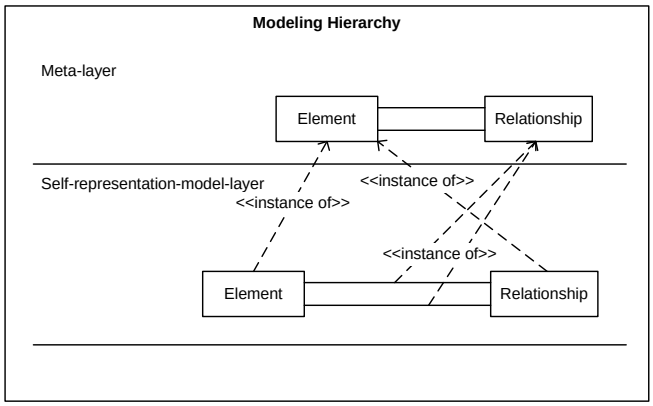


Figure 13 : Self-representation of metamodel

The instances of elements will have the same representation as elements in the metamodel. The instances of relationship are lines between two instances of elements. The fact that the instantiated relationship has another representation is not covered in this metamodel. As explained in the section 2.4 this is defined in the concrete syntax and mapped with the abstract syntax.

Next, we will consider the reflective ability of inspection and adaptation in more detail. For example, assume that a metamodel defines only a single inheritance construction. Through the self-adaptation, the inheritance construction can be changed in a multiple one. The ability to change the model is not always safe. For example, assume that the current metamodel already defines a multiple inheritance construction and through the self-adaptation we will change it in a single inheritance construction. This will lead to conflicts if the multiple inheritance construction is already used in defining the metamodel. In order to avoid this we should have restrictions on the ability for changing the metamodel.

In the modeling hierarchy, we use a recursive-top-layer for avoiding that another top-layer is needed. The recursive-top-layer does contain a metamodel that can represent itself. For creating the self-representation of the metamodel, we can build a virtual meta-layer that can produce the self-representation, as shown in Figure 14. The virtual-meta-layer does contain the same metamodel as the meta-recursive-top-layer. Based on this metamodel we can describe the complete metamodel below.

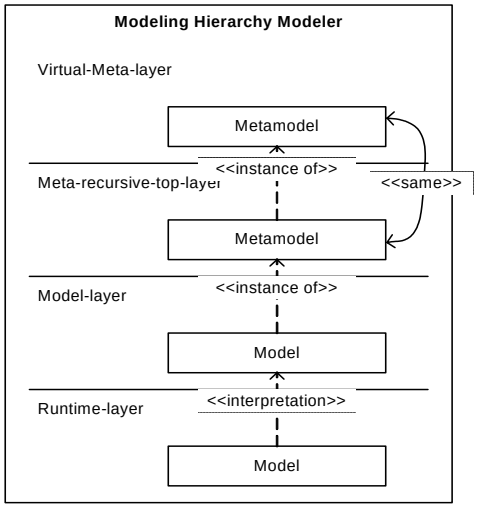


Figure 14 : Self-representation through a virtual meta-layer

2.9 Modularity

Modularity is the method for grouping and organizing models [21]. Modularity is one of the major architectural features in modeling, and is used very often. We can design a system as a set of fine-grained groups, where each group contains the elements that logically belong to each other. A well defined modularly architecture helps in understanding the system and simplifies the reuse of particular parts. Using the organized groups in combination with the ability to define dependencies between them will allow building a coherent architecture. The dependencies can allow that models that are defined in one group to be reusable for other groups. This can be done for further specifying models or reusing the same models in multiple groups.

We will illustrate the use of modularity in more detail according to a more concrete example. We define a grouped hierarchy for a business metamodel. The business metamodel contains a products, a sales, a purchase, and a warehouse group. The products group is divided into simple products and advanced products, which are nested groups inside the products group. This will have as result the grouped hierarchy as shown in Figure 15.

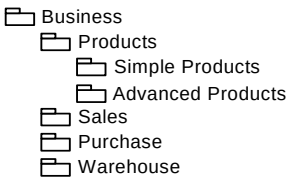


Figure 15 : Tree hierarchy business metamodel

The advanced products are specializations on the simple products. Furthermore, the sales, purchase, and warehouse groups will use the advanced product in these groups. For showing these dependencies, we introduce relationships, which can be used between groups. The purpose of the dependency is defined as label on the relationship. The dependencies between the business groups are shown in Figure 16.

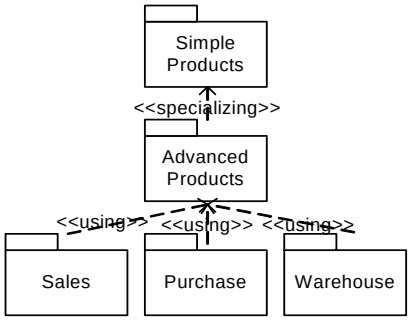


Figure 16 : Dependencies in business metamodel

3

MOF 2.0 Standard

The MOF 2.0 standard is described in the MOF 2.0 Core specification [10], which describes the abstract syntax and the semantics of the MOF. The MOF 2.0 Core specification is a metamodel that represents the MOF modeling language.

The MOF specification is written for programmers who build tools compliant with the MOF standard. This makes the specification in some point hard to read. With this section we aim to explain in more detail how to use the MOF standard. The basis of this section is extracted from the MOF specification, and enlarges it with our own interpretation on this subject. Therefore, we use where possible examples that directly show how to use a particular concepts of the standard.

The structure of the chapter is as follow, we will briefly describe the history of MOF. After that, the architecture used in the metamodel is considered, and the last section discusses the specification approach used to define the MOF 2.0.

3.1 History of MOF

The OMG, founded in 1989, has as its goal to provide a solution for reducing complexity, lower cost, and hasten the introduction of new software applications. Therefore, OMG introduced an architectural framework with supporting detailed interface specification. The specification should lead to interoperable, reusable, portable, software components based on standard object-oriented interfaces. The members of the OMG believe that the object-oriented approach to software construction best supports their goals. The first solutions were introduced as Object Management Architecture (OMA) and Common Object Request Broker Architecture (CORBA) [15].

The OMA provides a framework for implementing distributed systems. It defines four components, one of them is the Object Request Broker (ORB), which describes how objects interact in a distributed environment. Another component is the Object Services, which is used for general object management. Therefore, it performs tasks such as creating objects, access control, keeping track of relocated objects, etc. The other two, Facilities and Application Objects, support the end user with functions for many application domains, available through a class interface. CORBA is a concrete implementation of the architectures and specifications for the OMA ORB [15].

In 1996 the first request is released for a Meta-Object Facility. According to this RFP the MOF should define [13]: "*the interfaces and sequencing semantics needed to create, store and manipulate object schemas that define the structure, meaning, and behavior of other objects within the OMG Object Management Architecture.*"

In the same period the OMG releases a concept of modeling. This started in 1995 with a request for proposal for Unified Modeling Language (UML), and with the release of UML 1.0 in 1997 the first step was taken. This language can be used for specifying, constructing and documenting the artifacts of systems, and is very rapidly accepted in the world of software engineering

In 2001 the OMG launched the Model Driven Architecture (MDA) as the new approach for developing software using models and modeling techniques. The aim of the MDA is to reach an abstraction level that more concentrates on defining the structure and behavior of the system independent from the underlying implementation technology. The MDA adopted the already defined technologies for enabling the model-driven approach, as UML and MOF [4].

With the adoption of the MOF by the MDA, the MOF made a shift from being part OMA to being the core element of the MDA. This shift change the purpose of the MOF from being a framework to create, store and manipulate object schemas into being a meta metamodel used for defining metamodels, like UML.

In 2004 the architecture of UML is changed into an UML 2.0 superstructure and an UML 2.0 infrastructure library. In the same year the MOF 2.0 is released, with as major goal a better alignment with the UML. Therefore, the MOF 2.0 is reusing the UML 2.0 infrastructure library, which provides a better alignment between both standards.

In appendix A, we provide an overview of relevant adopted OMG standards and there releases in time.

3.2 Language Architecture

The MOF is used in combination with a metamodel hierarchy. In this section we will discuss the architecture of the metamodel hierarchy as well as the MOF architecture.

3.2.1 Metamodel Hierarchy

The MOF is designed to be the top layer in the OMG metamodel hierarchy. The metamodel hierarchy provided by the OMG contains four layers, namely:

- M3; meta-metamodel layer
- M2; metamodel layer
- M1; model layer
- M0; run-time layer

The M3 and M2 layers are *language specification layers*. The purpose of metamodels in M3 is to specify other metamodels. This layer contains only one metamodel, which is the MOF. The next language specification layer is the metamodel layer. The metamodels in this layer are more specific with respect to the meta- metamodel layer, but still abstract. The metamodels are used in specifying models. This layer can contain multiple metamodels. The model layer is a user specification layer. This layer will contain a concrete definition of the data. The run-time layer contains the objects instantiated out of the model. The representation of this is dynamic in time. In Figure 17 the OMGs metamodel hierarchy is given.

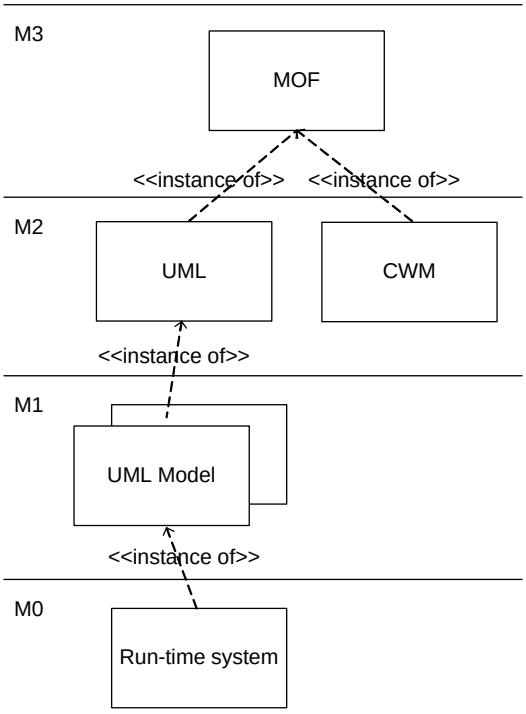


Figure 17 : OMGs metamodel hierarchy

As shown in the Figure 17, the M2 layer contains the UML and CWM metamodels, which are standardized metamodels based on the MOF provided by the OMG.

The MOF is a recursive top layer, as described in section 2.8, so in theory no additional language is needed to describe MOF. Besides describing itself, MOF is capable of describing metamodels in the M2 layer. All described metamodels in the M2 layer will have a strict *instance-of* relationship with the MOF metamodel. This will hold for the self-description of the MOF as well. The same relationship is used for defining the dependencies between the M2 and M1 layer, and the M1 and M0 layer.

3.2.2 MOF Architecture

As already discussed in section 2.9, a modular architecture is an essential part in modeling. This holds for the MOF as well. The whole structure that is used to define the MOF heavily relies on a modular architecture. The modularity for the MOF is introduced in the UML infrastructure library, as *"a principle of strong cohesion and loose coupling is applied to group constructs into packages and organize features into metaclasses"* [6].

In order to group constructs the MOF uses the concept of *package*. The packages do allow nested packages, which provide the ability for creating a hierarchy out of packages. When defining the package hierarchy used for the MOF, the outermost packages are the MOF and UML infrastructure library.

The dependencies between the MOF and UML infrastructure library are twofold, as shown in Figure 18. The first dependency is a reuse relationship, which is done for aligning reasons. The second dependency is the fact that UML is an instance of MOF.

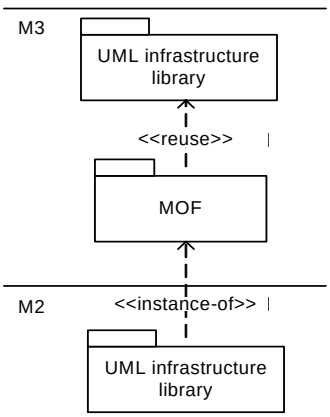


Figure 18 : Dependencies MOF and UML infrastructure library

In the next sections, we will further explain the packages that are in the MOF package hierarchy. We will first start with explaining the UML infrastructure library package and then continue with the MOF package. For the sub architecture of both packages the dependencies import and merge are used, in section 3.3.2 these dependencies are further explained.

3.2.3 UML infrastructure library

The purpose of the UML infrastructure library is defining common metalanguage elements, which can be reused for defining other metamodels, like MOF and UML [6]. The advantage of reusing the same infrastructure between more metamodels is to obtain architecturally aligned metamodels. The acrchitecturally alignment is obtained because of; both MOF and UML are using the same language elements as core language elements, which ensure that both architectures are equal.

The package hierarchy of the UML infrastructure library is shown in Figure 19; in the figure, the package hierarchy is limited to two levels.

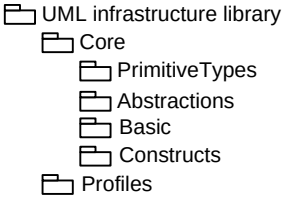


Figure 19 : Tree hierarchy UML infrastructure library

Core

The first subpackage of the UML infrastructure library is the Core with as subpackages PrimitiveTypes, Abstractions, Basic, and Constructs. The dependencies between these packages are shown in Figure 20. The dependencies between the Basic and Abstraction, and the Constructs and Abstraction are based on the subpackages of Abstractions, in the section below we will explain this in more detail.

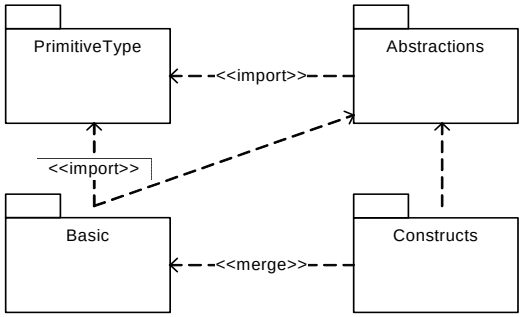


Figure 20 : Core package with dependencies between them

The Abstractions, Basic, and Constructs packages contain common metalanguage elements in such a way that each package defines a complete set for a metalanguage. The differences between these packages are the degree of abstractions. Furthermore, the PrimitiveTypes package contains predefined types that directly or indirectly are reused by the other packages.

- *Abstranctions*

  The Abstractions package contains the most abstract set of common metalanguage elements. It is further divided into a number of finer-grained packages that together form the complete set needed for a metalanguage. The abstract character and finer-grained packages structure is meant to support a high level of reuse in defining new metamodels. The Abstractions package itself is directly reusing the predefined types in the PrimitiveTypes package.

- *Basic*

  The Basic package provides a minimal set needed for a class-based metalanguage. The Basic package is not divided into sub-packages, as its intention is to be completely reused. For defining the Basic package a couple of packages from the Abstractions package are reused, and the PrimitiveTypes package is reused.

- *Constructs*

  The last package that the Core package contains is the Constructs package, which contains the most concrete set of common metalanguage elements. The purpose of these elements is to be used in object-oriented modeling. For defining the Constructs package the Basic package and parts of the Abstractions package are reused. Because, the complete Basic package is reused the PrimitiveTypes package is indirect reused as well.

The Constructs package is an example of showing the reusability of the Abstractions packages. As shown in Figure 21, the Constructs package merged almost the complete set of Abstractions subpackages.

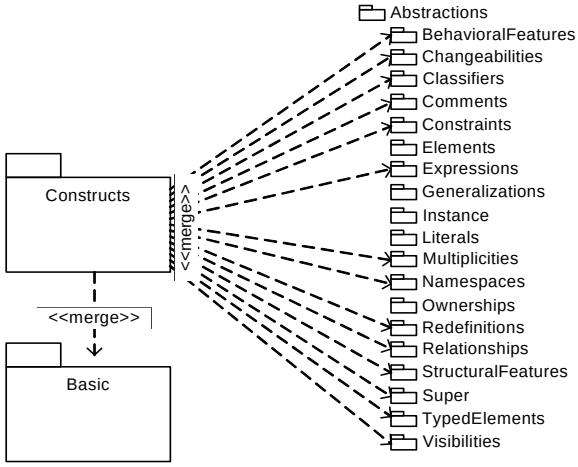


Figure 21 : Dependencies Constructs package

The Basic package reuses the Abstractions packages in a more lightweight manner, compared to the Constructs package. The Basic package import a set of four Abstractions subpackages, as shown in Figure 22.

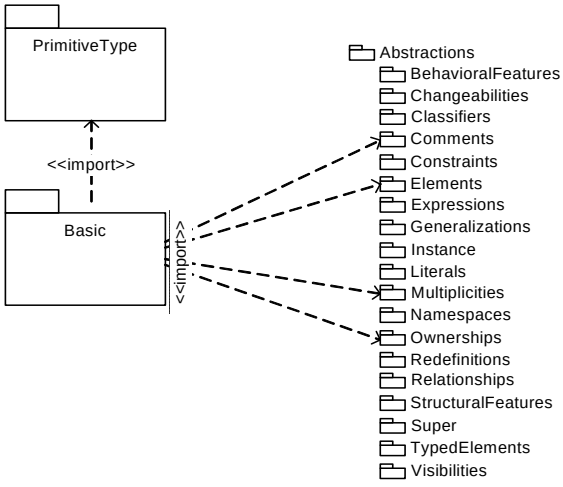


Figure 22 : Dependencies Basic package

Profiles

The second subpackage of the UML infrastructure library, the Profiles package, is used as mechanism to tailor existing metamodels towards specific platforms or domains. The package is dependent on the Core::Constructs package and aligned with the Extension package, used in the MOF. The major difference between these two is that the Profiles package uses a more lightweight approach with restrictions that are enforced to ensure that the implementation and usage of profiles should be straightforward and more easily supported by tool vendors [6]. The instance models of the MOF use the Profiles package, whereas the Extension package is part of the MOF.

3.2.4 MOF

Besides the UML infrastructure library as outermost package, the MOF package hierarchy contains the MOF package. The package hierarchy of the MOF is as shown in Figure 23.

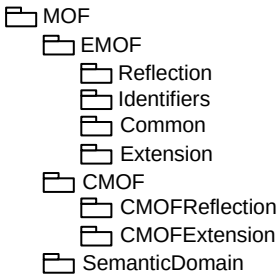


Figure 23 : Tree package hierarchy MOF

The MOF package contains three subpackages, namely the Essential MOF (EMOF), the Complete MOF (CMOF), and the SemanticDomain. The MOF defines the purpose for both EMOF and CMOF packages as: *"A primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions (by the usual class extension mechanim in MOF) for more sophisticated metamodeling using CMOF"* [10].

Both CMOF and EMOF packages are further divided into subpackages. These subpackages are stated as additional language capabilities for discovering, manipulating, identifying, and extending metamodels.

The MOF categorizes the following capabilities [10]:

- *Reflection*: Extends a model with the ability to be self-describing.
- *Identifiers*: Provides an extension for uniquely identifying metamodel objects without relying on model data that may be subject to change.
- *Extension*: A simple means for extending model elements with name/value pairs.

All the capabilities are introduced in the EMOF as packages. Each capability corresponds with a package, with exception of the capability Identifiers that is divided into an Identifiers package and a Common package. The CMOF will not introduce new capabilities, but only extend the capabilities introduced in the EMOF.

EMOF

A complete overview of the dependencies relationships of EMOF with other packages from the MOF package hierarchy is shown in Figure 24. The EMOF package merges the Core::Basic package and imports the Core::PrimitiveTypes package out of UML infrastructure library. Furthermore, the EMOF merges the introduced packages Reflection, Identifiers, Common, and Extension.

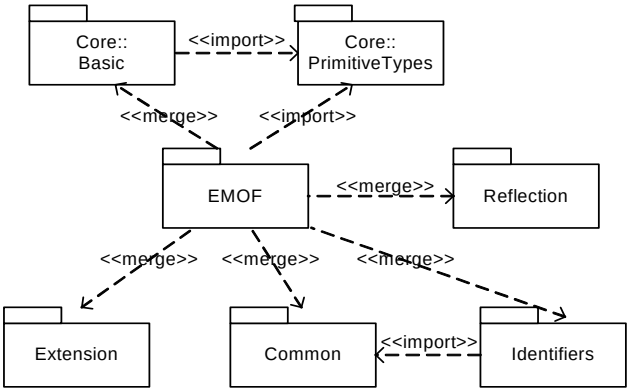


Figure 24 : Dependencies EMOF

- *Reflection*

The purpose of the Reflection package is to provide models with the ability for self-describing. For self-describing a model, we need the ability to create new elements. Creating new elements in EMOF will be done through instantiating meta-classes. For instantiating meta-classes, the Reflection package has the Factory capability, which contains operations for creating new elements as instances of a meta-class.

Furthermore, the Reflection package introduces the Object capability, which contains operations to manipulate and change properties. This will provide the ability to set the newly created elements with the correct properties values.

- *Identifiers*

The Identifiers package is an extension for uniquely identifying metamodel objects without relying on model data that may be subject to change. Providing the uniqueness is done with an identifier. The identifier is used to distinguish elements from each other.

Because of the reflection ability, the model data can be changed, and manipulated through itself. Providing the identifiers independently of the model data will avoid that these are changed and manipulated through reflection.

The identifier provides the identification as Extent and as URIExtent. The URI is the universally unique identification of the package following the IETF URI specification (RFC 2396). The identifier will import the Common package to maintain the identification for elements.

- *Common*

The Common package provides the ability for accessing multiple elements in such a way that the ordering is maintained and the uniqueness of each element is guaranteed. This package contains two elements, namely ReflectiveCollection and ReflectiveSequence. The ReflectiveCollection can be used for unordered collections and the ReflectiveSequence for ordered collections. The ReflectiveSequence will use an index for maintaining the ordering.

- *Extension*

The extension package will provide the ability to add tags to elements. These tags can contain named values, which are useful for adding information missing from the model.

CMOF

The second subpackage of the MOF package, CMOF, depends on other packages from the MOF package hierarchy as shown in Figure 25. The CMOF package merges the Core::Constructs package out of the UML infrastructure library and EMOF package and includes some additional language capabilities beyond the EMOF ones. These are defined in the CMOFReflection and CMOFExtension capability packages, which are merged as well.

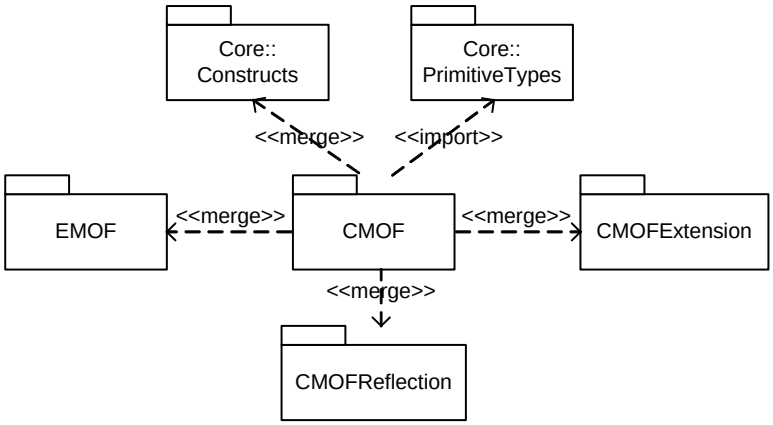


Figure 25 : Dependencies CMOF

The introduced packages of CMOF are an extension on the Reflection and Extension package of EMOF. The CMOFReflection package is introducing the ability for instantiating Association. As the EMOF already introduced the instantiation of classes, the instantiation of associations is done in the same way.

SemanticDomain

The third subpackage of the MOF package is the SemanticDomain package. The SemanticDomain package describes the semantic domain for the CMOF and is optional for the EMOF. According to [10] the SemanticDomain package describes: *"the functional capabilities of a modeled system and how those capabilities are related to elements in the model"*. Therefore, the CMOF is taken from being a metamodel to being a modeled system (CMOF model representing a CMOF system). So according to this the semantic domain does describe the way MOF elements are instantiated. More precisely, the semantic domain is purely concentrated on class diagrams from the Core::Constructs package, and is an extension on the abstract syntax of the Core::Abstractions::Instances package, as shown in Figure 26.

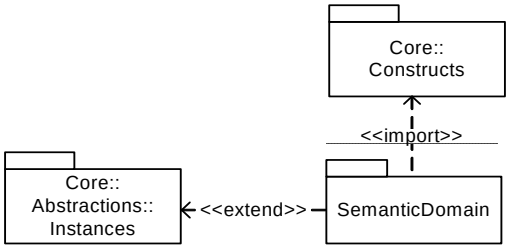


Figure 26 : Dependencies SemanticDomain

For illustrating the SemanticDomain package in more detail we will use Figure 27. which represents the complete overview of all packages related to the SemanticDomain, and also shows the dependencies between the packages. The representation is based on the in section 2.4 defined representation of a metamodel with correspondent dependencies.

The CMOF metamodel is a combination of the package CMOF and SemanticDomain. The CMOF package as described above is the abstract syntax part of the metamodel. The SemanticDomain package is a CMOF model representing a CMOF system. The SemanticDomain package is the semantic domain part of the metamodel. This semantic domain will provide the meaning for each element defined in the CMOF package. Using a semantic mapping, we can combine the CMOF package with the SemanticDomain package.

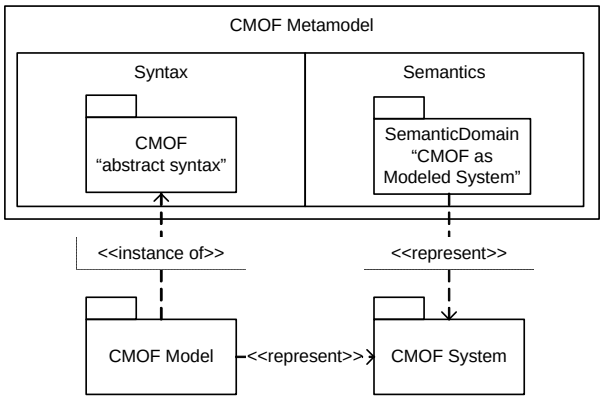


Figure 27 : CMOF as modeled system

3.3 Language Formalism

This chapter will cover the specification approach used for the MOF. First, we will briefly discuss the specification approach used in the MOF. Second, the fundamental MOF techniques are mentioned in more detail, as import, merge and instantiation.

3.3.1 MOF specification

To explain what is used to describe the MOF, we will provide an overview of all specification techniques involved in describing the MOF.

The MOF specification is stated to use the following specifications techniques [10]: *"a subset of UML, an object constraint language, and precise natural language"*. The subset of UML is the UML infrastructure library that is extended by the MOF.

As described in section 3.2.1 the MOF is located in the top-layer. Therefore, in the modeling hierarchy there is no meta-layer above the M3 layer. For defining the dependencies between the additional specification techniques and the MOF we will use a virtual-layer above the top-layer, which contains all additional specification techniques. The virtual-layer with the specification techniques used for the MOF is shown in Figure 28.

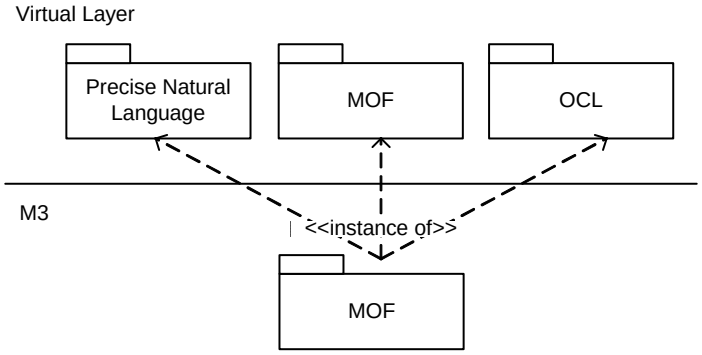


Figure 28 : Virtual layer for MOF specification

First, the MOF specification is not self-contained, meaning that we need other specifications as well to reach a complete specification of the MOF. In the case of the MOF, the *UML specification: infrastructure* [6] is needed as well, to reach a complete MOF specification. The UML specification: infrastructure does contain the, in section 3.2.3 already mentioned, UML infrastructure library. For both specifications, the same specification techniques are used, which will allow a better reuse of one specification with the other specification.

According to [6], the specification techniques are used in gain of the following goals:

- *Correctness.* The specification techniques should improve the correctness of the metamodel by helping to validate it. For example, the well-formedness rules should help validate the abstract syntax and help identify errors.
- *Precision.* The specification techniques should increase the precision of both the syntax and semantics. The precision should be sufficient so that there is no syntactic nor semantic ambiguity for either implementors or users.
- *Conciseness.* The specification techniques should be parsimonious, so that the precise syntax and semantics are defined without superfluous detail.
- *Consistency.* The specification techniques should complement the metamodeling approach by adding essential detail in a consistent manner.
- *Understandability.* While increasing the precision and conciseness, the specification techniques should also improve the readability of the specification. For this reason a less than strict formalism is applied, since a strict formalism formal techniques.

As shown in the Figure 28, the MOF is located in the virtual-layer, instead of the UML infrastructure library (subset of UML). The UML infrastructure library is not used to describe the MOF, but is extended by the MOF. Therefore, the UML infrastructure library and MOF are located in the same layer. The MOF can describe the UML infrastructure library and the MOF itself. For this reason, the MOF is a specification technique for the MOF, and is located in the virtual layer as well.

Additional specification techniques

We will first explain the additional specification techniques that are used in the virtual layer in more detail.

Precise natural language

Precise natural language is capable to describe all additional constructs needed. The advantage of the precise natural language is the fact that it is easy to read and write. The drawback is that even precise natural language is not very precise, and will result in ambiguities. Especially semantics defined in natural language will have the well-known limitations, as described by Harel and Rumpe [22].

In most cases, the additional constructs defined in precise natural language are for explaining concepts in more detail. When we need to define concepts that are essential, it makes sense to define it in a formal language as well. Formal languages have as advantage to be unambiguous. In contrast to the precise natural language, formal languages are mostly hard to read and write, because of the heavy use of mathematics for making the language unambiguous.

Object Constraint Language

The MOF specification states to use an object constraint language, more precise this is the Object Constraint Language (OCL). The OCL as described in [20], is a formal language used to describe expressions on models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Avoiding that the language is only readable and writeable for mathematical persons, the language is designed to have an easy to read and write character while remaining unambiguous.

For illustrating the use of OCL in the MOF specification, we will use the following examples. The first example is the use of OCL in the UML infrastructure library. The example shows an extract from the UML infrastructure library that defines a constraint and an additional operation for a particular element.

The Core::Abstractions::Multiplicities::MultiplicityElement (MultiplicityElement) contains an attribute lower, which can not be a negative integer literal. Therefore, the MultiplicityElement contains the constraint [6, page 67]:

> **Constraints**
> [2] The lower bound must be a non-negative integer literal.
> `lowerBound()->notEmpty() implies lowerBound() >= 0`

The lowerBound() operation is an additional operation of the MultiplicityElement, defined as [6, page 67]:

> **Additional Operations**
> [4] The query lowerBound() returns the lower bound of the multiplicity as an integer.
> `MultiplicityElement::lowerBound() : [Integer];`
> `lowerBound = if lower->notEmpty() then lower else 1 endif`

In this example the constraint and additional operation is written in the OCL. The constraint is an example of an invariant condition that must hold for the system being modeled. The additional operation is a described query that can be used over objects described in a model.

The second example we give, represents the use of OCL in the MOF::SemanticDomain package. In the MOF::SemanticDomain package, the OCL is used to define the capabilities. The example shows an Object Capability, defined as [10, page 64]:

> **Object Capabilities**
> `Object::container(): Object modeled as Instance::container(): ClassInstance`
> `post: result = self.get(self.owningProperty())`

The owningProperty() operation is an additional operation, defined as [10, page 71]:

Additional Operations

[4] This returns the single Property that represents the current owner of the Object based on current instance values; may be null for top level objects

```
Object::owningProperty(): Property
post: result = self.allProperties->select(op| op.isComposite and self.get(op) <>
null)
```

Both Object Capability and Additional Operation are described as queries in the OCL.

MOF specification

We will now continue with explaining the structure of the MOF specification. We will divide the specification into the language aspects as introduced in section 2.3.

Concrete Syntax

The concrete syntax of the MOF can be divided into two parts, namely the part that describes the concrete syntax used for representing the MOF and the concrete syntax used for representing the MOF based metamodels. In the case of the MOF, for both the same concrete syntax is used.

A subset of UML is used as concrete syntax. The subset of UML covers the notation for class diagrams, defined as the UML infrastructure library. UML infrastructure library contains the complete notation for class diagrams needed for defining the MOF graphical notation. In the UML infrastructure library each element can contain a description in precise natural language that defines the graphical notation according to the class diagram.

Illustrating the graphical notation in more detail we will use the following examples. The first example, as shown below, is the description for a Classifier as located in the UML infrastructure library Core::Abstractions package [6, page 37].

Notation

The default notation for a classifier is a solid-outline rectangle containing the classifier's name, and optionally with compartments separated by horizontal lines containing features or other members of the classifier. The specific type of classifier can be shown in guillemets above the name. Some specializations of Classifier have their own distinct notations.

The second example will show the standard specification for a textual notation. The UML infrastructure library uses a variant of the Backus-Naur Form (BNF) to specify the legal formats [6], as shown in the example below [6, page 68]:

Notation

<multiplicity> ::= <multiplicity-range>
<multiplicity-range> ::= [<lower> '..'] <upper>
<lower> ::= <integer>
<upper> ::= '' | <unlimited_natural>*

The example shows the BNF specification for the notation of Multiplicity. According to this specification notations in the form of [0..5] or [0..*] are valid Multiplicity notations.

The EMOF and CMOF do not introduce any new notations. Therefore, the complete notation of these two can be extracted from the UML infrastructure library. In the appendix C a detailed overview of the notation for the EMOF and CMOF is given.

Abstract syntax

The MOF specification is mostly concentrated on defining the abstract syntax. The abstract syntax for the UML infrastructure library, EMOF, and CMOF are all defined in a similar way. The abstract syntax is

defined as class diagram. Each element in the class diagram contains a detailed description that explains the element in more detail.

For describing the class diagram the MOF specification is used, by means of the self-describing. This will be further explained in section 3.4. The detailed description for each element in a class diagram is described in precise natural language and OCL.

We will now show a fragment from the MOF specification as example how the combinations of specification techniques are used to form the abstract syntax of the MOF. The following Figure 29 is a copy out of the MOF specification [10, page 31, 33], which describes the Identifier package, as part of the EMOF.

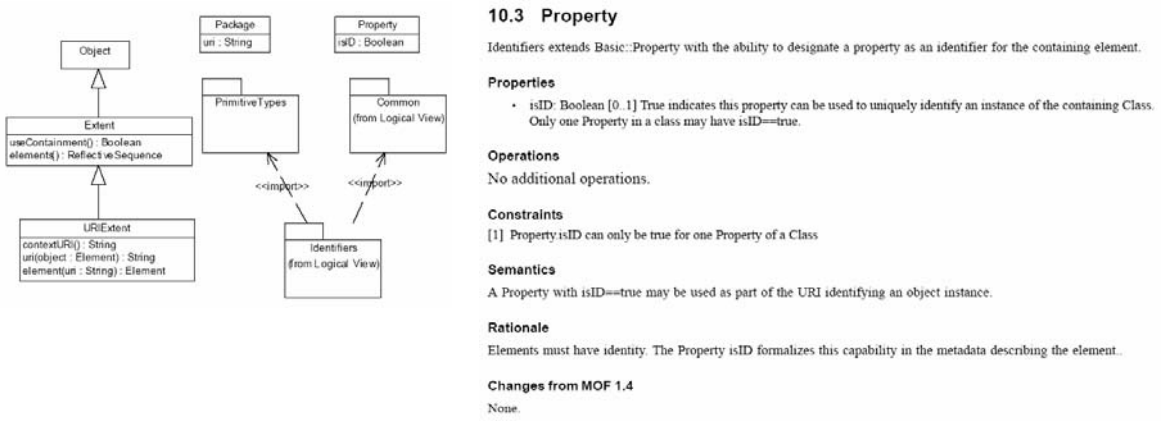


Figure 29 : MOF abstract syntax example

The left part of Figure 29 is the abstract syntax of the Identifier capability package. This is defined in a UML notation. For further and additional explanation, each element has a more precise description. The additional explanation is build out of a couple of subsections, these subsections are:

- Description: the common description for the element.
- Properties: description for each property.
- Operations: description for each operation.
- Constraints: set of rules that must be satisfied by all instances of this element.
- Semantics: the meaning of the element.
- Rationale: the reason why an element is introduced.
- Changes from MOF 1.4: description of the changes made compared with MOF 1.4.

The additional explanations are written in precise natural language; with exception of the constraints part, which can be defined in OCL as well.

Semantic Domain

The semantic domain of the MOF does contain the instances model. Besides the instances model it contains capabilities and additional operations. The instances model is a class diagram described in MOF. The capabilities and additional operations are expressed in OCL.

The capabilities defined in the SemanticDomain will map the operations defined in the abstract syntax with ones modeled in the instances model. This is shown in the example below, where the Object::container() operation is mapped with the Instance::container() operation, where the Object::container() operation is part of the abstract syntax and the Instance::container() operation is part of the instances model.

> **Capabilities**
> **Object::container(): Object** modeled as **Instance::container(): ClassInstance**

In the example (Figure 29) used for showing the representation of the abstract syntax of the MOF, each element in a class diagram can contain a semantics description. These textual descriptions can be used to

interpret the meaning of each element when involving in instantiating. The semantics can contain references to elements that are defined in the instances model of the semantic domain.

For example, the semantic description of the Class as located in the UML infrastructure library Core::Basic. The semantic description for the Class is shown below [6, page 97]:

> **Semantics**
> … The instances of a class are objects. … An object has a slot for each of its class's direct and inherited attributes. …

In this case, the slot is an element in the instances model.

3.3.2 Import and Merge

As discussed before, for defining the MOF a modular architecture is used. For providing the ability to build a modular architecture, we will need constructions that support the combining of fine-grained packages. The constructions used through the MOF for combining packages are also described by the MOF, according to self-describing ability of the MOF. These constructions are import and merge.

The import and merge constructions will allow the ability to build a modular architecture, wherein each modular defined part can be reused. Both import and merge are reusing modular parts such that the reused parts stays connected with it source, as a causal connection. So when the source is changed the reused parts will change as well.

Import

There are two possible ways for importing, namely with elements and packages. The package import can be seen as importing each element in the package using element import.

The element import provides the ability to use the element in the importing package. The element import works through reference. The importing package obtains a reference element of the elements in the imported package, as shown in Figure 30. In the importing package we can further define the reference element, but the element in the imported package will always stay the same.

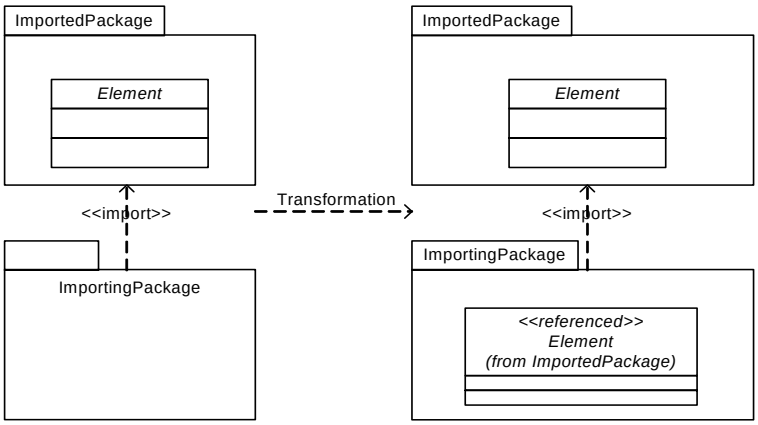


Figure 30 : Transformation of packages according import

Merge

Merging is a more advanced technique than importing. The technique allows predefined constructs to be combined with each other. The merge can only be between packages, where one will be merged with the other one. If in both packages an element represents the same entity, the elements will be merged into a single element. Important is to define when an entity is equal to another entity. In examples we will always define entities equal if names are the same. Normally, entities are identified by there direct reference but for readability, we will use name reference.

In merging we always have a merged package and a receiving package. After the merge, a conceptual package is created with the package merge as result. So after defining the merge relation between two packages a transformation can be performed that will create the actually resulting package, as shown in Figure 31 [6].

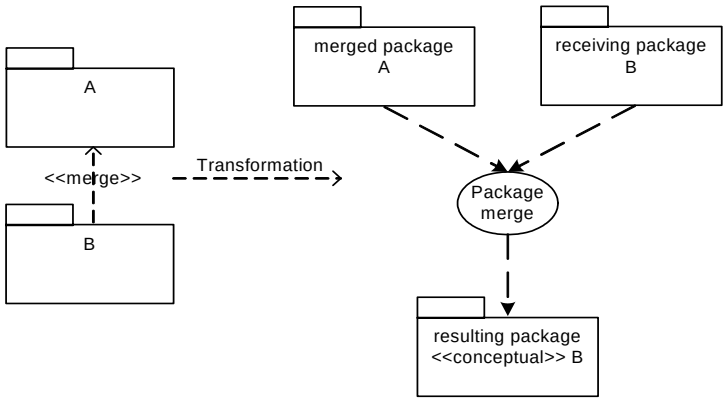


Figure 31 : Transformation of packages according merge

The exact way of merging is defined in the rules of package merge, which are part of the semantics. These rules are divided into constraints and transformations rules. The constraints rules will be used to check if a merge can be performed correctly. On the other hand, the transformations rules will be used to define the actual effect of the merge. To perform a valid merge, each rule defined in the merge semantics should be applicable.

Example Merge and Import

We will now show the actual use of the merge and import, using a merge and import as performed in the UML infrastructure library. The example is shown in Figure 32. The Core::Abstractions::Ownerships package does contain a <<referenced>> *Element* and an *Element*. The <<referenced>> *Element* is imported from the Core::Abstractions::Elements package. By means of the *Element*, which has the <<referenced>> *Element* as superclass, an association is added. The introduction of *Element* is needed, because it is not possible to directly change the <<referenced>> *Element*. We can only refer to an imported element, as done with the superclass relationship.

The Core::Abstractions::Comments package has an import relationship with the Core::Abstractions::Ownerships package. Therefore, it contains a <<referenced>> *Element* that represents the imported *Element*. The Core::Abstractions::Comments package adds an *Element*, a Comment, and two associations.

Next, both Core::Abstractions::Comments and Core::Abstractions::Ownerships package are merged with the Core::Constructs package.

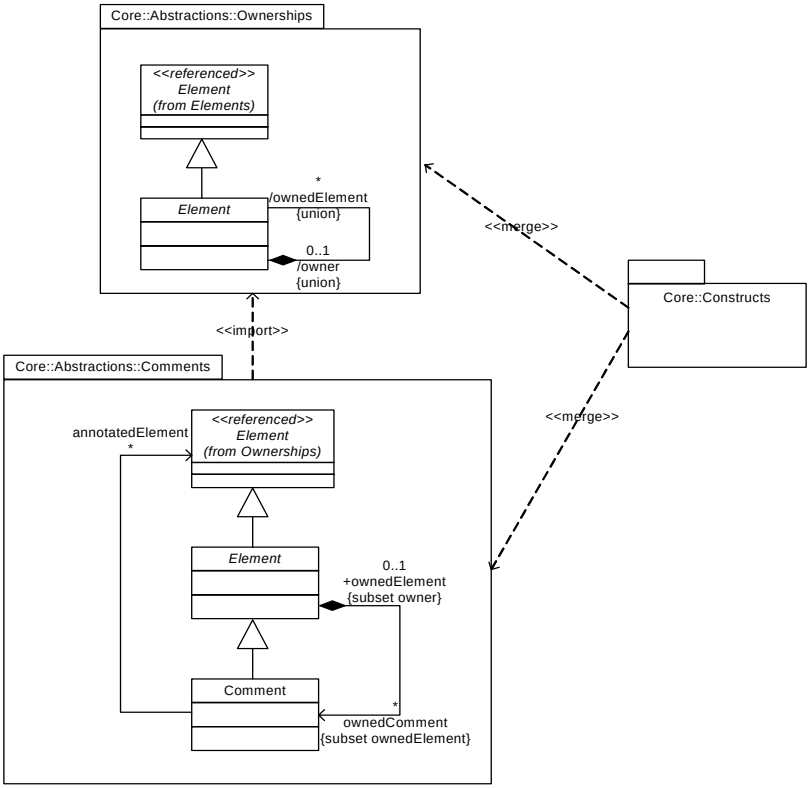


Figure 32 : Example merge and import

For the Core::Constructs package, we can draw a resulting package, which contains the elements after the merge is performed. The resulting package for Core::Constructs is shown in Figure 33. The elements that are merged should be equal. In the example elements are equal based on their names. So in Figure 32 the *Element* in Core::Abstractions::Ownerships package is equal to *Element* in Core::Abstractions::Comments package. Furthermore, the <<referenced>> *Element* in Core::Abstractions::Comments is representing the *Element* imported from Core::Abstractions::Ownerships package. These elements are reduced into one *Element*, which contains all associations. The <<referenced>> *Element* in Core::Abstractions::Ownerships package is not merged because; this element is representing the *Element* from Core::Abstractions::Elements package, which is not part of the merge.

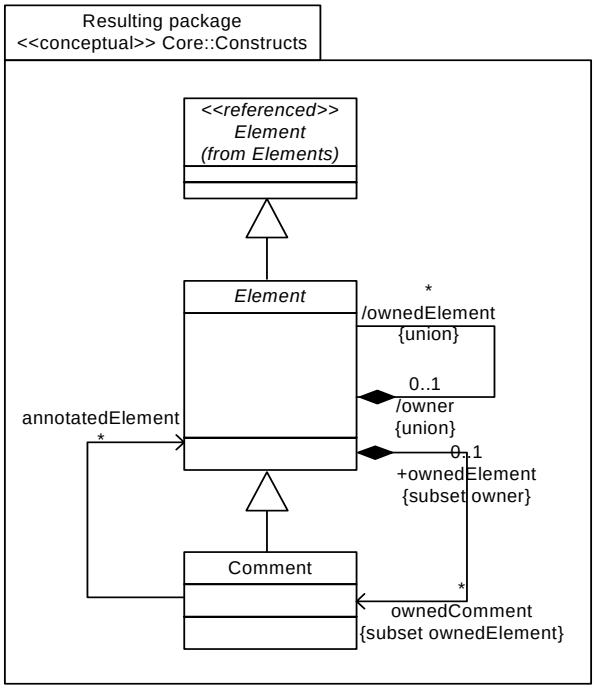


Figure 33 : Resulting package for Core::Constructs

3.3.3 Instantiation

We already discussed in section 3.2.4 that through the reflection capability of the MOF new elements can be introduced. In this section, we will discuss in more detail the instantiation behavior of each language construction. The instantiation behavior for each language construction is described in the correspondent semantics. Furthermore, the language construction can contain a number of constraints rules. The instantiation of a language construction will be correct if each rule holds.

Because most of the language constructs are directly reused from the UML infrastructure library, these semantics are reused as well. Inside the UML infrastructure library, semantics are as much as possible aligned with each other. With as result, that comparable elements used in EMOF and CMOF will have the same meaning.

For the notation, we are using the instances representation, which is used in object diagrams. This notation is described in the appendix C. Furthermore, the correspondent invoking sequence is shown in a dashed square.

Instantiating classes

The result of instantiating a class is an object. A class can only be directly instantiated if it is a non-abstract class. When instantiating the class each superclass is indirectly instantiated as well. The object will contain all properties and operations defined in its class and its superclass [6].

The capability for creating objects as instances of classes is introduced in the EMOF::Reflection package, in the class Factory. For realizing this, the Factory class contains a couple of operations. Besides that, it contains the attribute Factory.package. By means of this attribute, we can define the package that contains the classes that we need to instantiate. For example, if we need to create objects as instances of EMOF classes, we need to assign the EMOF package to an instance of the Factory, as shown in Figure 34. The assigning of EMOF package to an instance of the Factory is based on the reflection of the package.

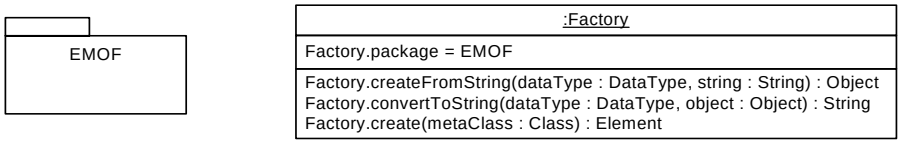


Figure 34 : :Factory with assigned EMOF package

The :Factory object can now create objects as instance of each non-abstract class in the EMOF package. For doing this, the Factory contains the operations createFromString and create. So when we, for example, invoke the create operation with parameter Class an object will be returned which is an instance of Class, as shown in Figure 35.

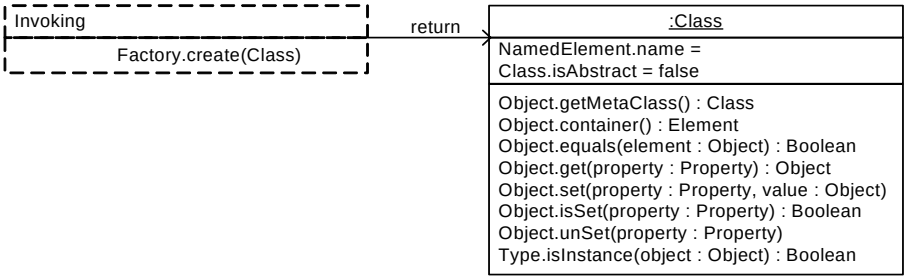


Figure 35 : Create operation of Factory

Because the Object class is in the inheritance tree of Class, the returned object given by the create operation its properties and operations as well. These operations will provide the ability to access, reason about, and alter the object. For example, with the Object.set operation we can set the NamedElement.name property with the value *Video*, which gives the object a name, as shown in Figure 36.

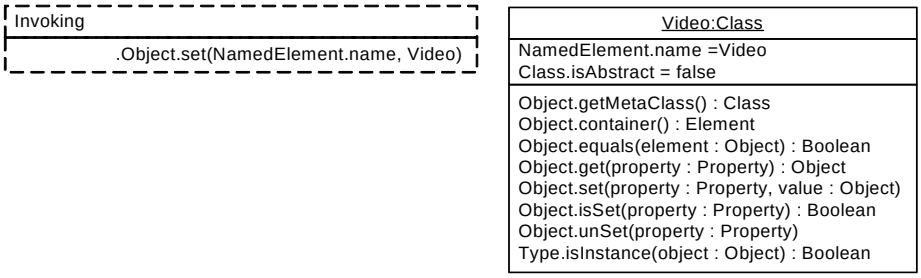


Figure 36 : Example for altering the object

The actually invocation of operations is not covered by the Reflection package in the EMOF, and there are no semantics for executing an operation in EMOF. The CMOF does introduce the ability to actually invoke an operation. For this purpose, CMOF's CMOFReflection package extends the Object class with the operation invoke. This invoke operation allows invocation of all operations that are part of the class. So for instance, we would like to invoke the operation Object.get for the property NamedElement.name, the Object.invoke operation is used as shown in Figure 37.

For adding parameters to the invoking operation, we can use the invoke parameter arguments. The parameter arguments is an Argument DataType, which is introduced in the CMOF's CMOFReflection package as well. The Argument DataType contains two attributes, name and value, for building a correct Argument. So, as shown in Figure 37, for the operation Object.get we need to set the parameter property with value NamedElement.name. Therefore, we define an argument with name property and value NamedElement.name.

When serving the invoke operation with the correct argument it returns the Element that is normally returned by the invoked operation. So in the case of the Object.get operation, the Object.invoke operation will return the Object with as value Video.

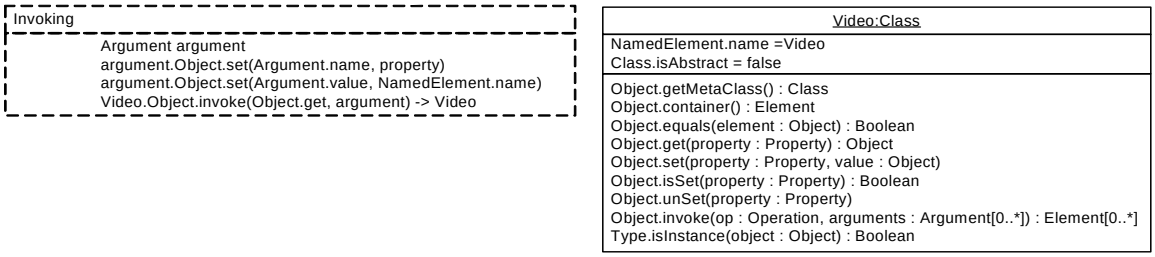


Figure 37 : Invocation of an operation

Operations, properties, and parameters

For operations, properties, and parameters it is also possible to have multiple instantiations. These elements handle the multiplicity in the same way. The multiplicity provides the ability to set lower and upper bound for denoting the number of possible instantiations. Furthermore, it is possible to assign if the instantiated elements should be in order and if each element has to be unique. The two attributes can be used in combination, which provides four possible states. In the following Table 1, each combination is worked out for a property.

{ordered, unique} OrderedSet	**{ordered} Sequence**
aProperty[0] = name	*aProperty[0] = name*
aProperty[1] = description	*aProperty[1] = name*
aProperty[2] = code	*aProperty[2] = code*
{unique} Set	**{} Bag**
aProperty[] = name	*aProperty[] = name*
aProperty[] = description	*aProperty[] = name*
aProperty[] = code	*aProperty[] = code*

Table 1 : Combinations for ordered and unique

In Table 1 the ordering is maintained by the index behind the property name. Further, the uniqueness is based on the name, so in a unique collection each name must be unique.

For maintaining the collection of multiple instantiated elements, the EMOF introduced the Common package. This package provides two classes, namely a ReflectiveCollection class for maintaining unordered multiple instantiated elements and a ReflectiveSequence for maintaining ordered multiple instantiated elements. These classes are used in combination with the operations used in maintaining the properties. For example, the Class can own multiple ordered attributes. These attributes can be set with the operations Object.set as shown before. Therefore, the operation Object.set will recognize that you would like to set a multiple ordered property and use the ReflectiveSequence to maintain them, as shown in the Figure 38, where the Video class is set with the properties name, description, and code.

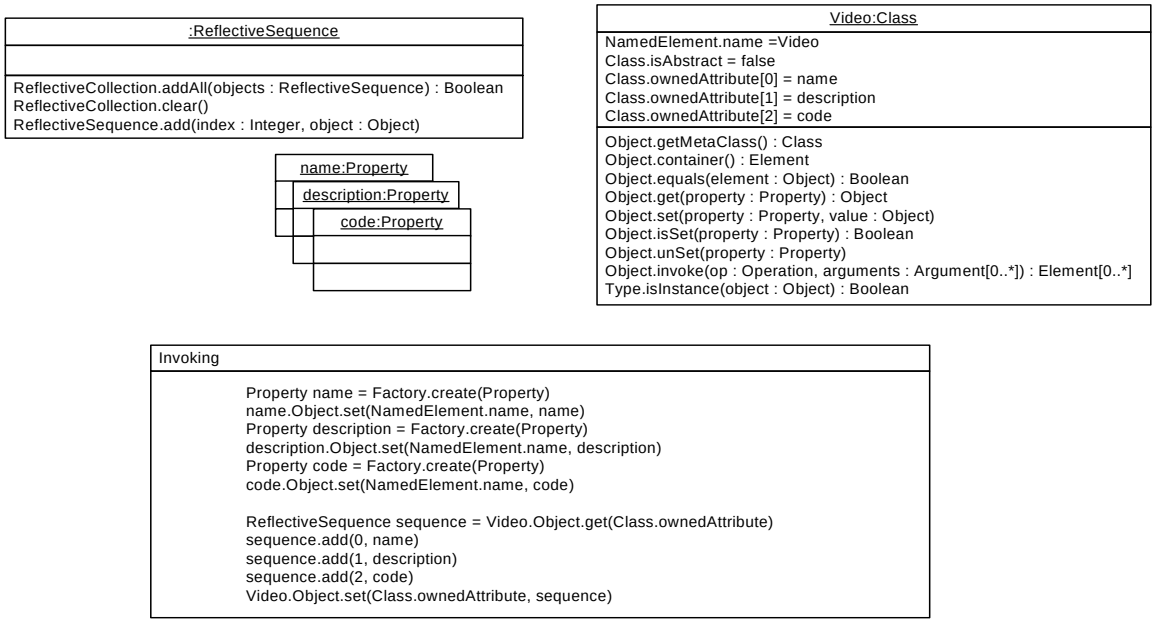


Figure 38 : A collection maintain with ReflectiveSequence

Both EMOF and CMOF are not directly supporting any facility for maintaining the uniqueness of each element in a collection of elements.

Associations

The EMOF only provides the ability to instantiate classes; in contrast, the CMOF provides the ability to instantiate associations as well. The instantiation of associations is introduced in CMOF's CMOFReflection package. This provides a Link class, which denotes an instantiated association. Furthermore, the Factory class is extended with the operation createLink, which will create a Link. An association is always between two classes, and a link is always between two objects. Therefore, the operation createLink will contain two parameters for setting both objects.

The ability to create links will allow us to enlarge the example shown in Figure 38 with links between the properties and class, as shown in Figure 39.

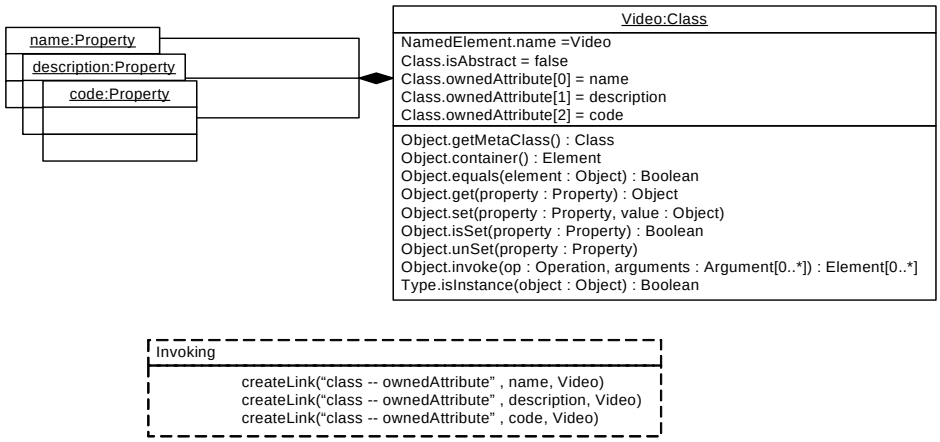


Figure 39 : The creation of links

In all the examples above, we use the names for referring to elements; this is done to make the example more readable. The actual referring is done by direct object reference, which is supported through the EMOF's Identifiers package. This package introduces two classes, which can be used for getting information

about element identifiers. The way an element becomes a member of the identifier capability is not covered in the MOF Core specification.

An instantiation only succeeds when all defined constraints' hold as well. The CMOF has the ability for assigning constraints to elements. Therefore, it introduces a Constraint class, which will contain the constraint, and an attribute for assigning elements for which the constraint should hold. For highlighting that a constraint does not hold, the CMOF introduced the Exception class in the CMOFReflection package. This Exception class can be used for providing information about the constraint that does not hold.

3.4 Self-representation

In this chapter, we will describe the self-representation ability of the MOF. The MOF stated the self-representation as follows [10]:

"EMOF and CMOF are both described using CMOF, which is also used to describe UML 2.0. EMOF is also completely described in EMOF by applying package import, and merge semantics from its CMOF description. As a result, EMOF and CMOF are described using themselves, and each is derived from, or reuses part of the InfrastructureLibrary."

For explaining the self-representation, we will consider the prominent constructions used to define the metamodel and instantiate these based on the metamodel itself. We will define this for both EMOF and CMOF.

Normally we define the abstract syntax of the MOF as a class diagram, but because the MOF is self-describing, we can show the abstract syntax also as an object diagram. For showing the self-representation of the MOF, we will instantiate parts of the MOF and represent them as object diagrams. Therefore, we will use the construction as shown in Figure 40, the top-layer is the MOF as class diagram and the layer below will be the MOF as object diagram. In the object diagram we can show the settings of the attributes, which are normally used in mapping the concrete syntax. In the representation of the MOF as object diagram the most important attributes will be highlighted.

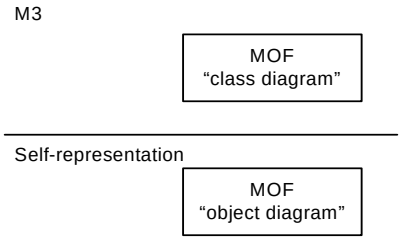


Figure 40 : Construction of showing the self-representation of the MOF

In all the examples, we will refer to classes by use of their names. The official way in referring to model elements is by direct object reference but for readability, we will use name reference.

3.4.1 Self-representation in EMOF

For building the self-representation for the EMOF, we will need the EMOF without the use of any package merge and import. The EMOF is not capable of defining those constructs. Based on the semantics of the merge and import construction, we can extract this construction out of the EMOF through actually performing the merge and import. Appendix B contains the complete view of EMOF after performing the merge and import.

Class

We will start with the self-representation of the Class. Class is a non-abstract class with a Boolean attribute isAbstract, which has a default value false, as shown in Figure 41.

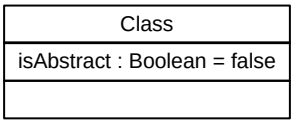


Figure 41 : EMOF Class

Class is an instance of the metaClass Class. The Class attributes will be set with "Class" as NamedElement.name and the Class.isAbstract will remain false. For creating the Boolean attribute isAbstract, a Property is instantiated and configured with NamedElement.name being isAbstract, Property.default being false, and TypedElement.type being Boolean, the other attributes are maintaining there default values. The Type Boolean, which is the value of TypedElement.type, is an instantiation of PrimitiveType, predefined in the EMOF::Core::PrimitiveType. Properties which have as type an indirect instantiation of DataType, will be notated as attributes inside classes, as in the case of the isAbstract:Property. The object diagram for the Class is shown in Figure 42.

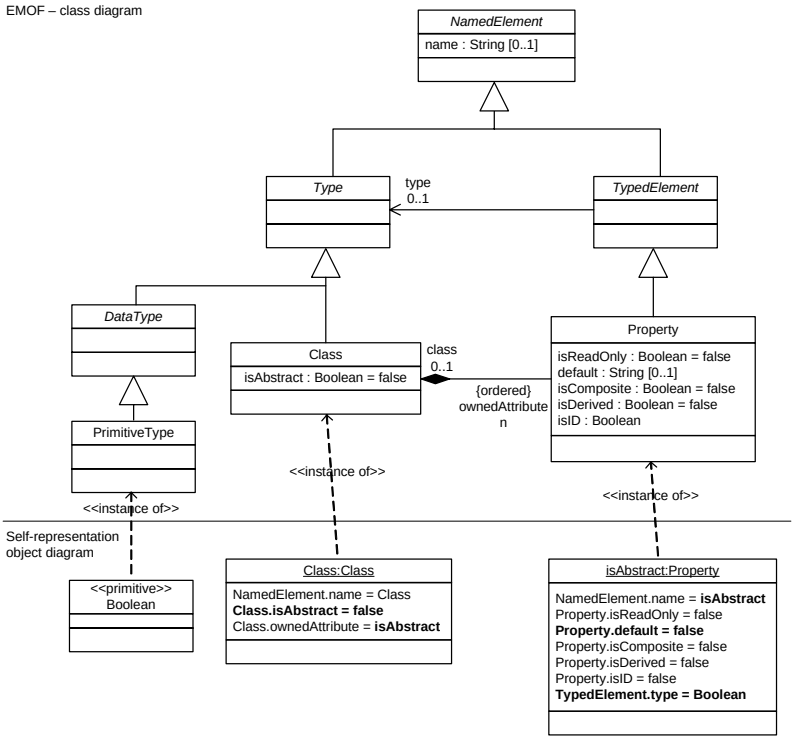


Figure 42 : Object diagram Class

Inheritance

The following construction we will consider is inheritance. We will take as example the inheritance relationship between Type and Class. In this case, the non-abstract class Class is inheriting from the abstract class Type, as shown in Figure 43.

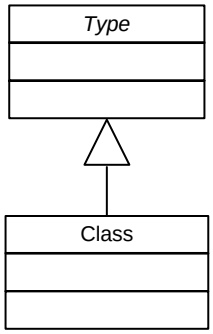


Figure 43 : Inheritance relationship between Type and Class

To create the element Type, Class is instantiated and the attributes NamedElement is set with "Type", further attribute Class.isAbstract has value "true". Class is instantiated as shown in the example before. Further, the association end Class.superClass has value "Type", which will denote that Class is inheriting form Type. The inheritance relationship between Type and Class is shown in Figure 44 as object diagram.

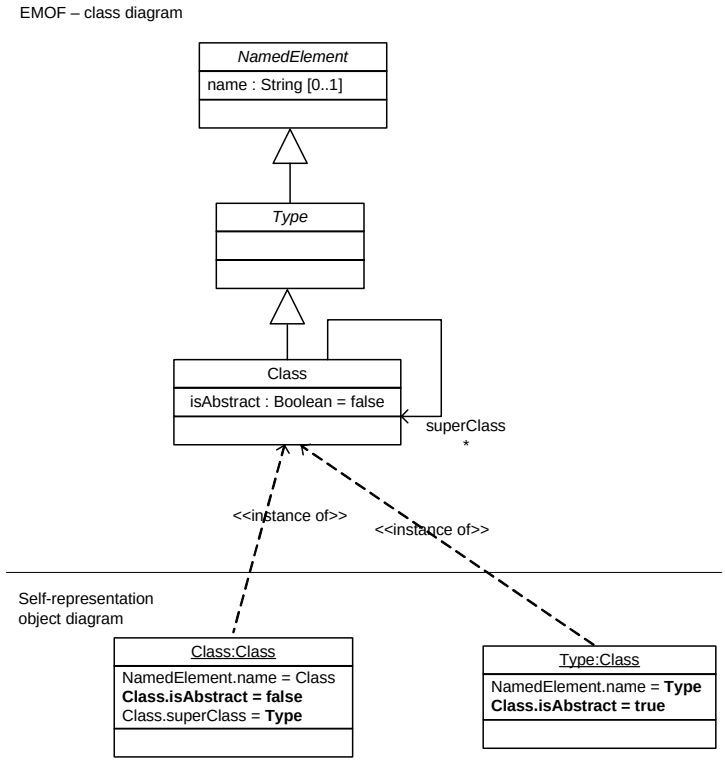


Figure 44 : Inheritance relationship between Type and Class as object diagram

Association

The EMOF defines associations as a couple of properties configured as each other's opposite. As example, we will describe the association between Class and Property. There is an association between both non-abstract classes, which has a composite end at the Class side. Further, the association contains the name "class" with MultiplicityElement lower bound 0 and upper bound 1, the name "ownedAttribute" with MultiplicityElement unlimited and constraint ordered. The example is shown in Figure 45.

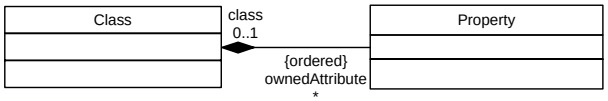


Figure 45 : Association between Class and Property

For constructing the association, we will instantiate two properties, attribute NamedElement of one property has value "ownedAttribute" and attribute NamedElement of the other property has value "class". For defining the composite end at the Class side the attribute Property.isComposite of ownedAttribute:Property has value "true". For defining that both properties are forming together an association the attribute Property.opposite of ownedAttribute:Property has value "class" and the attribute Property.opposite of has class:Property value "ownedAttribute".

Further, for ownedAttribute:Property the association end TypedElement.type has value "Property", the attribute MultiplicityElement.isOrdered has value "true", the attribute Multiplicity.isUnique has value "false", the attribute MultiplicityElement.lower has value "0", and the attribute MultiplicityElement.upper has value "*", other attributes remain their defaults. For the class:Property the association end TypedElement.type has value "Class", the attribute MultiplictyElement.isUnique has value "false", and the attribute MultiplicityElement.lower has value "0", other attributes remain their defaults. The association between Class and Property as object diagram is shown in Figure 46.

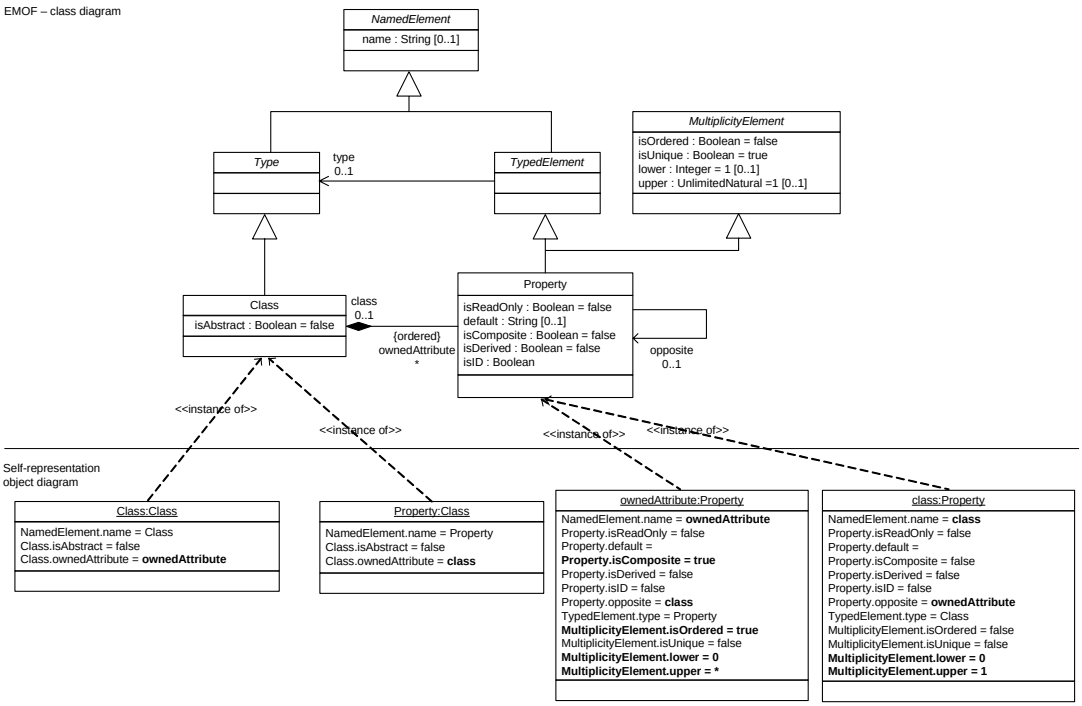


Figure 46 : Association between Class and Property as object diagram

The next example we will consider is an association with a navigated end. The example we use is Class that contains an association to itself for defining the superClass. The navigated end of the association is assigned with the name superClass with MultiplicityElement unlimited, as shown in Figure 47.

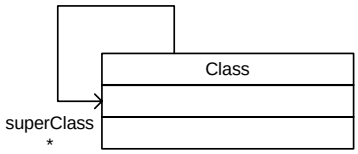


Figure 47 : Association with navigated end

Again, the Class is instantiated as shown in previous examples. For the association the superClass:Property is instantiated based on Property. Furthermore, for this object the attribute TypedElement.type has value "Class", the attribute MultiplicityElement.isUnique has value "false", the attribute MultiplicityElement.lower has value "0", and the attribute MultiplicityElement.upper has value "*", the other attributes remain their defaults. The instantiation of a navigating association is comparable with the instantiation of an attribute, with those differences that the TypedElement.type for the navigating association is set with an instance of Class and in the case of an attribute with an instance of Datatype. The navigated end represented as object diagram is shown in Figure 48.

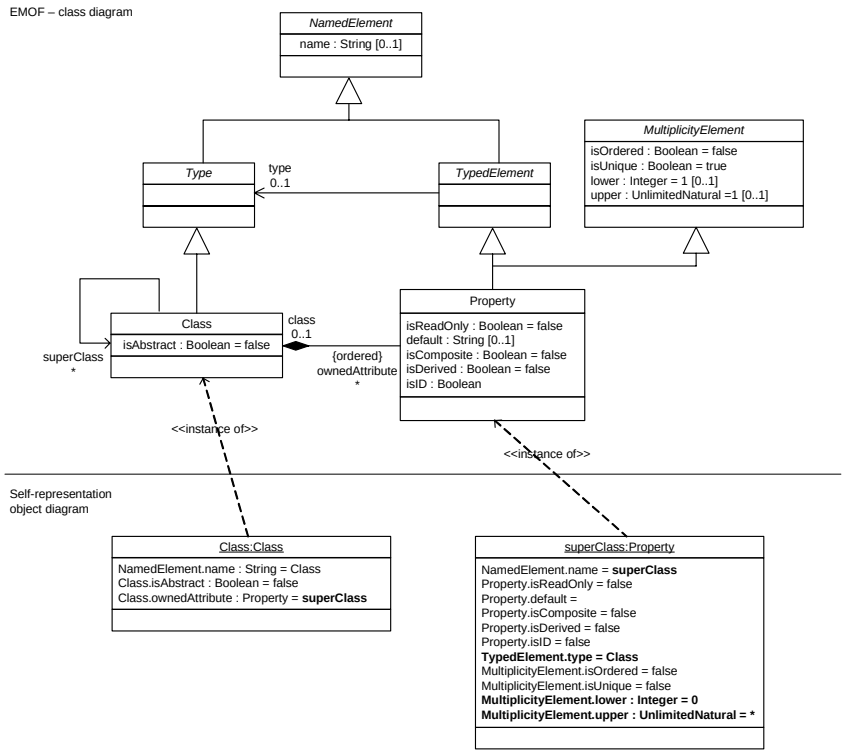


Figure 48 : Navigated end as object diagram

Operation

Besides defining properties, we can also define operations. For example, the abstract class Type contains an isInstance with as return type Boolean and as Parameter an element with type Element, as shown in Figure 49.

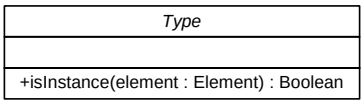


Figure 49 : Operation

An operation is a combination of the Operation class and the Parameter class. In the case of our example we have an element:Parameter instantiated from Parameter. For assigning the type for the element:Parameter as TypedElement.type, we need to instantiate an Element class as well. The isInstance:Operation is instantiated based on Operation, and assigned with the element:Parameter as value for ownedParameter attribute. Furthermore, the isInstance:Operation has the type Boolean, assigned as TypedElement.type. In order, the Class.ownedOperation attribute is assigned with isInstance, which will provide the Type:Class with the operation. The instantiation is shown in Figure 50.

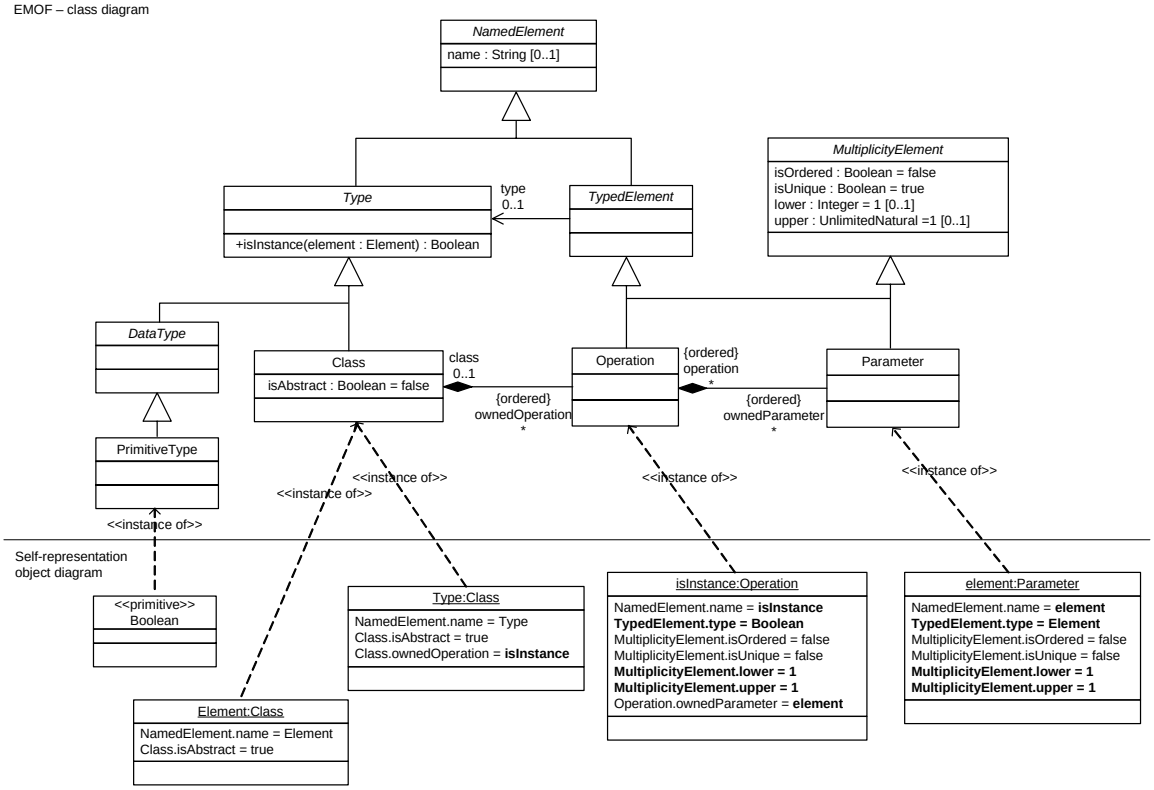


Figure 50 : Operation as object diagram

The operation contains a + sign as well. This + sign denotes that the operation is public, which is part of the visibility. The EMOF does not contain any constructions for defining visibility, as described in [10] chapter 12.4 EMOF Constraints; *"[4] Core::Basic and EMOF does not support visibility. All property visibilities expressed in the UML MOF model will be ignored (and everything assumed to be public). Name classes through names thus exposed should be avoided."*

3.4.2 Self-representation in CMOF

The CMOF defines on top of the EMOF some more advanced constructions. The EMOF contains the capability to instantiate classes. The CMOF enlarges its capability with the possibility to instantiate associations as well. According to this capability, we can use the instantiated associations in the object diagram as well. Furthermore, the CMOF provides additional capability according to defining associations. In the section below, we will show the CMOF possibilities according to associations in more detail.

Associations

Associations in EMOF are coupled properties with an opposite attribute pointing to each other. The CMOF introduced the Association class, which contains the properties that represent the association. Besides that, the CMOF is capable of instantiating an Association as a Link. For explaining the associations in CMOF in more detail, we will first take a same example as used in EMOF. Figure 51 shows the example.

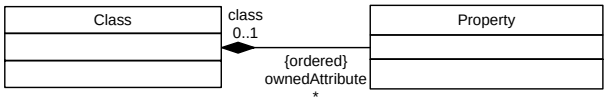


Figure 51 : Association between Class and Property

For building the association in CMOF, as done in EMOF two properties can be instantiated and set with the correct name and multiplicity. For combining the two properties, we can now use the instance of the Association class. The two properties are associated with the association through the instantiation of the association association/memberEnd. In the example the association end "class" is composite. Therefore, the attribute Property.isComposite of ownedAttribute:Property has value "true". The association between Class and Property as object diagram is shown in Figure 52.

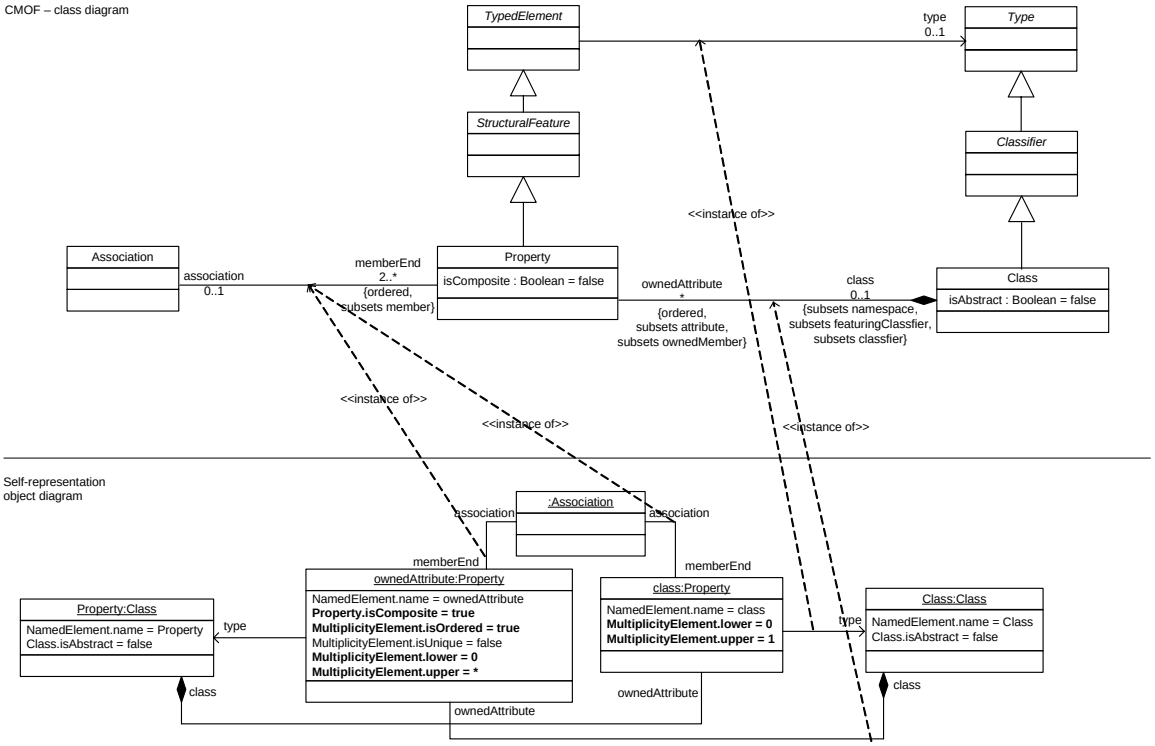


Figure 52 : Association between Class and Property as object diagram

As shown in the Figure 52, the associations in the class diagram also contain subset constraints. The ability for defining subsets is first introduced in the CMOF. The subset constructions provide the ability to define

association ends in a collection. For showing how to instantiate subset constructions, we will use the following example, as shown in Figure 53.

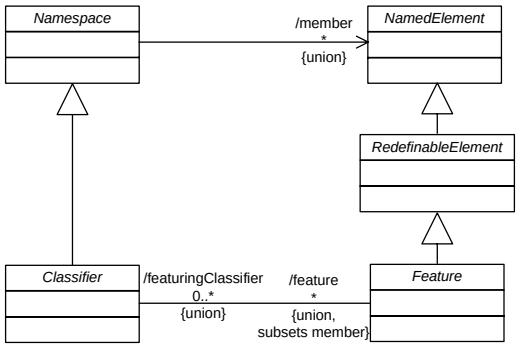


Figure 53 : Subset of association end

In the example, association end feature is a subset of member. Therefore, the association end member does contain a union constraint, which denotes that the associating end is a derived union of its subsets. The complete instantiation of the example is shown in Figure 54. The subset relationship is realized by setting member as the subsettedProperty for feature. The object diagram of the example contains two instantiated associations, for the association with the navigated end the instance of relationship is shown in Figure 52. The instance of relationship of the other association is comparable with the instantiated association in Figure 54.

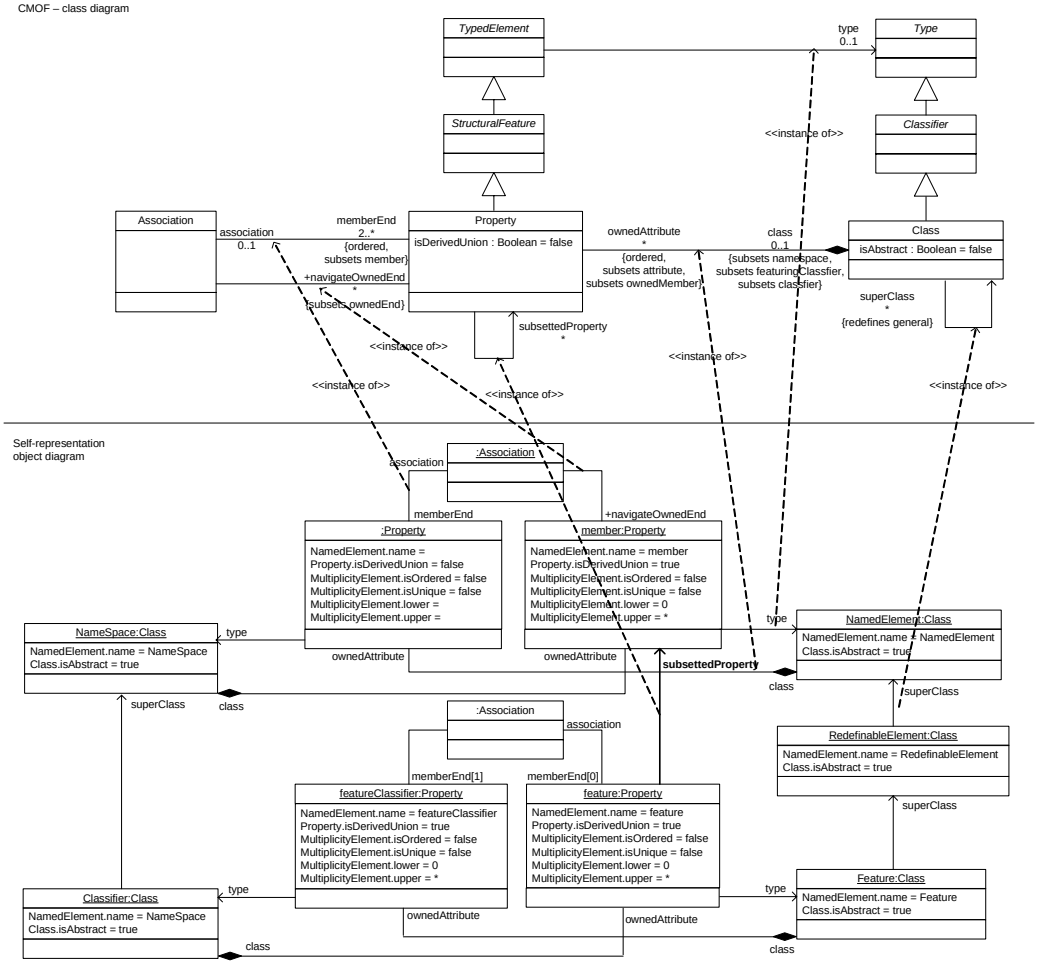


Figure 54 : Subset of association end as object diagram

Another ability for association end is to redefine an inherited association end. This is done in the case of Class, the association superClass is redefining the association general, as shown in Figure 55.

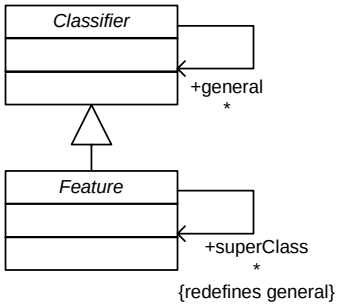


Figure 55 : Redefine association end

The redefining of association ends work in the same way as with subsets, with the differences that the association redefinedProperty is used between properties, as shown in Figure 56.

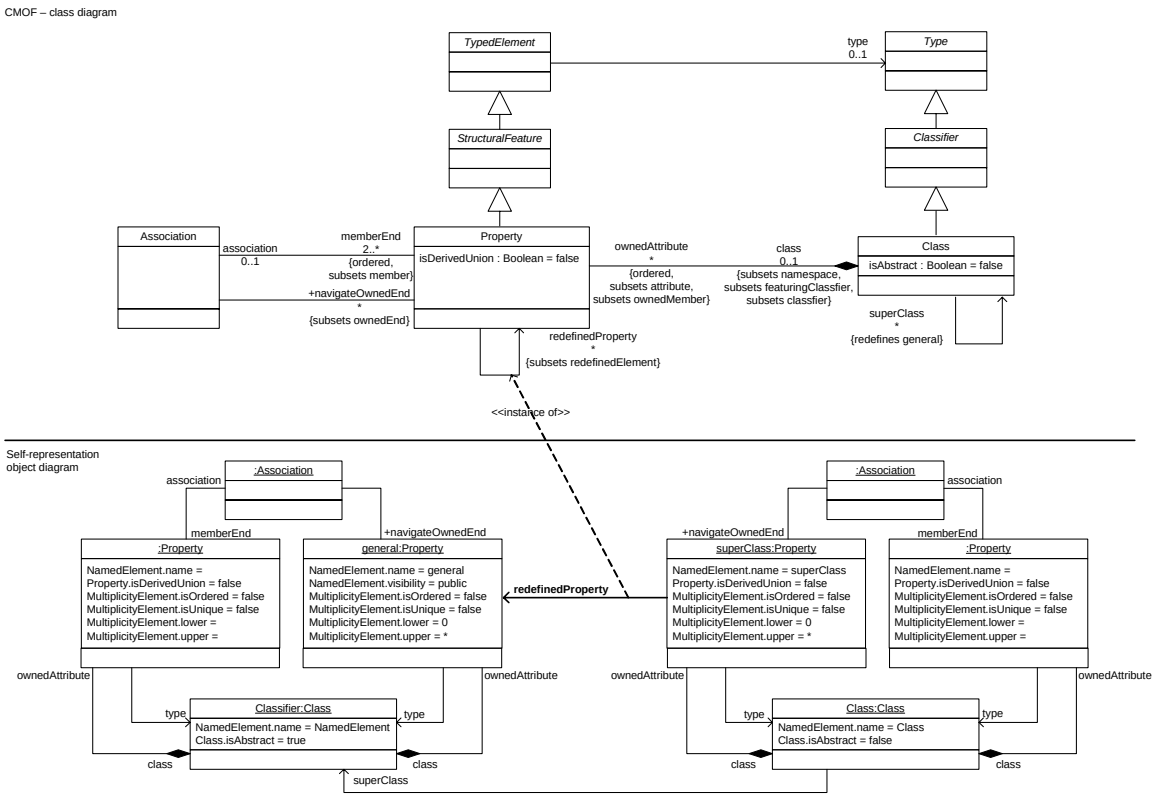


Figure 56 : Redefine association end as object diagram

4

MOF based example

For investigating how the MOF actually can be used, we will develop an example metamodel based on the MOF. Most examples now available for the MOF are in the area of UML. The examples mostly concentrate on extending MOF constructions in more advanced ones.

Instead, we would like to provide a more concrete example, which is easy to understand, even without knowledge about the MOF specification.

As example, we will build a metamodel for electrical diagrams. We will take a small subset that covers a limited area of the electrical diagrams domain. In the example, we will try to provide information for both the model as well as the run time layer, so that we can cover the whole four layered metamodel hierarchy used with the MOF.

4.1 Electrical diagram

The electrical diagram we will use, can contain the following components: resistors, switches, and a DC voltage source. A resistor has a resistance in ohms (Ω). A switch can have two states: on or off. The DC voltage source has a voltage in volts. The components are connected with each other using wires. The combination of components and wires forms an electrical circuit. Through the electrical circuit a current in amperes is flowing. Figure 57 contains two electrical circuits that are implemented based on the electrical diagram definition.

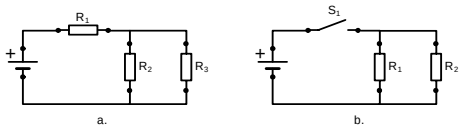


Figure 57 : Examples of electrical circuits

Furthermore, a couple of operations can be applied on an electrical circuit. We can apply the Ohm's law for calculating the voltage over a resistor or a combination of resistors. Ohm's law is:

$$U=I*R$$

where 'U' is the voltage in volts, 'I' is the current in amperes, and 'R' is the resistance in ohms. The equivalent resistance (R_{eq}) of a combination of resistors can be calculated through the following properties:

Calculating serial resistors: $R_{eq}=R_1+R_2+\ldots+R_n$

Calculating parallel resistors: $\frac{1}{R_{eq}}=\frac{1}{R_1}+\frac{1}{R_2}+\ldots+\frac{1}{R_n}$

The named R's, like 'R_1', in the calculation corresponds with the resistors in the electrical circuits. The serial and parallel calculation can be used in combination. For example, for the electrical circuit in Figure 57 a. the calculation for equivalent resistance over resistors R_1, R_2, and R_3 will be:

$$R_{eq} = R_1 + \frac{1}{\frac{1}{R_2} + \frac{1}{R_3}}$$

4.2 MOF hierarchy

The Electrical Diagram metamodel is built as metamodel based on the MOF. The metamodel provides the structure for building the electrical diagrams as models and the ability to execute them. In Figure 58 the Electrical Diagram Metamodel is placed in the MOF layered hierarchy.

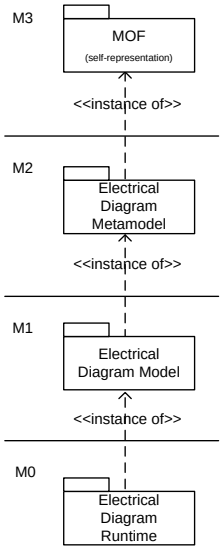


Figure 58 : Modeling hierarchy for Electrical Diagram metamodel

For defining the Electrical Diagram Metamodel we will use the same style as used for UML infrastructure library and MOF itself. Therefore, the abstract syntax of the Electrical Diagram metamodel will be given as a class diagram, together with a detailed description. As in the UML infrastructure library each defined element can contain a description about the notation. We will do this also to define the concrete syntax for the Electrical Diagram metamodel.

In the electrical diagram, new components are introduced. We will explain these components based on the well-defined and well-understood concepts in the mathematics, like Ohm's law and the function for calculating the equivalent resistance.

4.3 Electrical Diagram Metamodel

We build the Electrical Diagram Metamodel based on the CMOF. This done because the CMOF provides more modeling constructions that can be used in building the metamodel, like the merge and import of packages and the subsetting of associations.

In Figure 59, the package structure of the Electrical Diagram Metamodel is given.

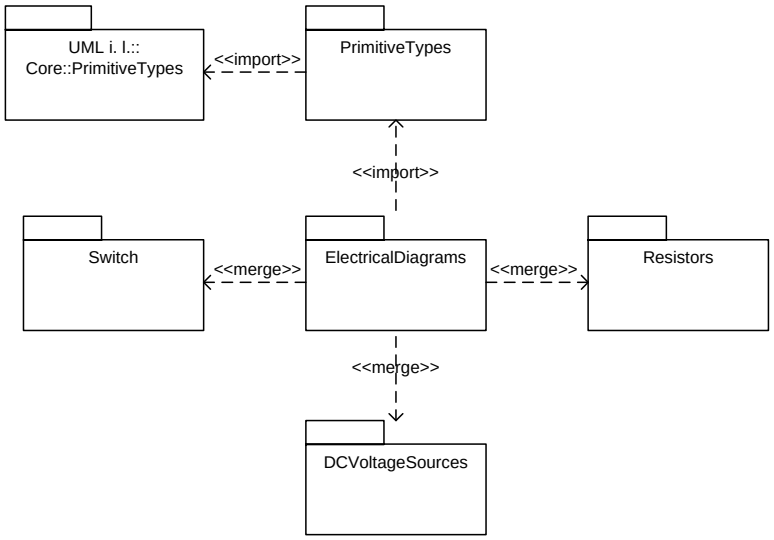
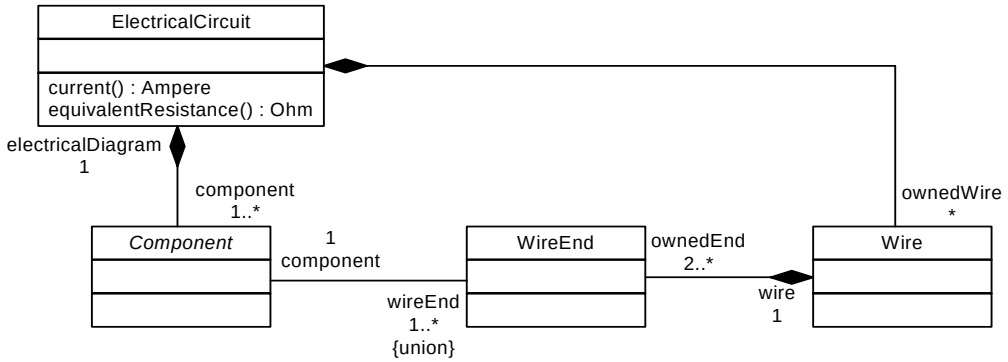


Figure 59 : Package structure Electrical Diagram Metamodel

4.3.1 ElectricalDiagrams



ElectricalCircuit

Description:
The ElectricalCircuit Class is representing an electrical circuit. The Class has as capability calculating the equivalent resistance and the flowing current in the electrical circuit.

Associations:

• component 1..*	Components owned by the ElectricalCircuit
• ownedWire *	Wires owned by the ElectricalCircuit

Operations:

• current() : Ampere	Returns the current in Ampere that is flowing in the circuit.
• equivalentResistance() : Ohm	Return the equivalent resistance in Ohm of the circuit.

Component

Description:
Component is an abstract classes for generalize the elements that can be add to the electrical circuit.

Associations:

- ElectricalCircuit 1 — ElectricalCircuit where the Component is a member of.
- wireEnd 1..* — The WireEnds owned by the Component.

WireEnd

Description:
An instance of a WireEnd is the connection between a component and a Wire.

Associations:

- component 1 — The Component owned by the WireEnd.
- wire 1 — The Wire owned by the Component.

Wire

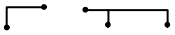
Description:
The Wire is connecting components with each other, which is based on the owned WireEnds.

Associations:

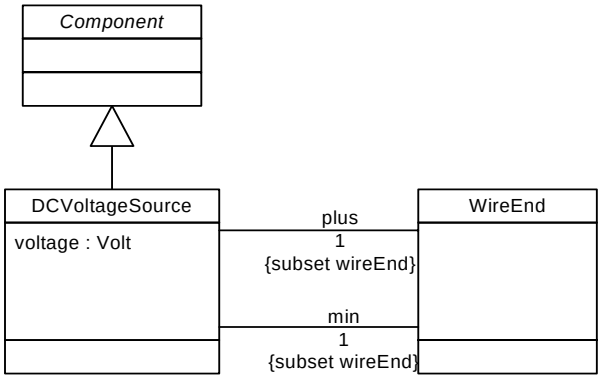
- ownedEnd 2..* — The WireEnds owned by the Wire. At least two WireEnds are needed to create a Wire.

Notation:
The wire is a line between components. The wire ends of the wire are dots.

Example:



4.3.2 DCVoltageSource



DCVoltageSource

Description:
The DCVoltageSource is the voltage source for the circuit, which provides a direct current voltage.

Attributes:

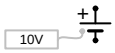
- voltage : Volt — The voltage value of the DCVoltageSource in Volt.

Association:

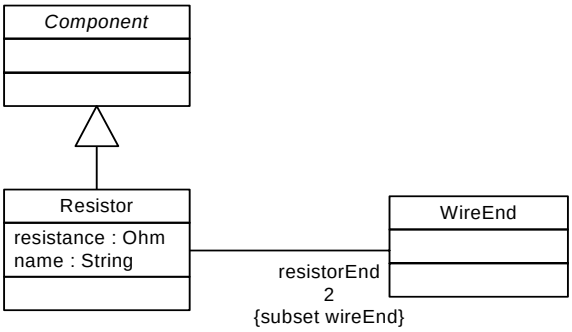
- plus 1 — The plus WireEnd owned by the DCVoltageSource, which subsetting the wireEnd as is owned by Component.
- min 1 — The min WireEnd owned by the DCVoltageSource, which subsetting the wireEnd as is owned by the Component.

Notation:
The DC voltage source are two lines at a right angle on a wire. The plus wire end of the DC voltage source is denoted with a thin line and a "+" sign. The min wire end of the DC voltage source is denoted with a thick line. The voltage value is added through a gray label.

Example:



4.3.3 Resistors



Resistor

Description:
Component that can be used as a resistor in the circuit, which will be identifying by its name and providing a certain amount of resistance.

Attributes:

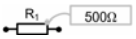
- resistance : Ohm — The amount of resistance in Ohm contained by the Resistor.
- name : String — The name contained by the Resistor, which is for identifying the Resistor in the ElectricalCircuit.

Associations:

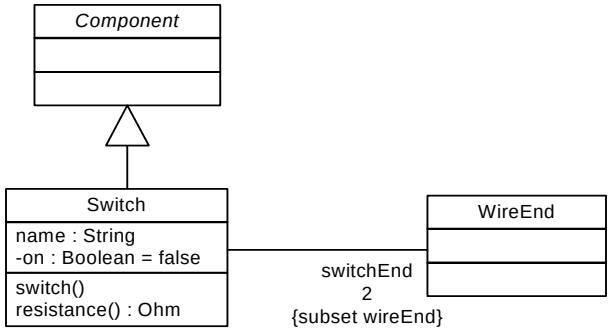
- resistorEnd 2 — The WireEnds owned by the Resistor, which subsetting the wireEnd as is owned by the Component.

Notation:
The resistor is a rectangle with two wire ends. The resistor contains the name, and the resistance value is added through a gray label.

Example:



4.3.4 Switch



Switch

Description:
Component that can be used as a switch in the circuit, which will be identified by its name. Furthermore, the Switch can have two states, namely an on and off state. Depending on the state the switch will have a zero or infinity resistance value.

Attributes:

- name : String — The name contained by the Switch, which is for identifying the Switch in the ElectricalCircuit.
- -on : Boolean = false — Private attribute used for controlling the state of the switch, the default state of the switch is off.

Associations:

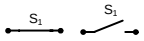
- switchEnd 2 — The WireEnds owned by the Switch, which subsetting the wireEnd as is owned by the Component.

Operations:

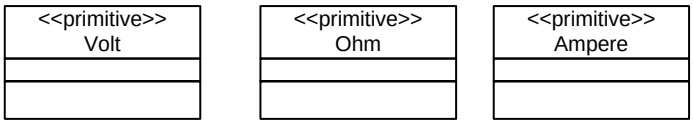
- switch() — Operation will change the state of the switch.
- resistance() : Ohm — Return the resistance value in Ohm of the switch.

Notation:
The switch is a line that can have two positions, a horizontal line is a switch in the on state, and a slope line is a switch in the off state. Furthermore, it contains two wire ends.

Example:



4.3.5 PrimitiveTypes



Volt

Description:
Volt is a primitive type representing the voltage value. An instance of volt is an element in the (infinite) set of Reals.

Notation:
The notation for volt is the capital letter V.

Ohm

Description:
An ohm is a primitive type representing the resistance value. An instance of ohm is an element in the (infinite) set of Reals.

Notation:
Ohm will appear as the type using the Greek capital letter omega (Ω) as notation.

Ampere

Description:
An ampere is a primitive type representing the current value. An instance of ampere is an element in the (infinite) set of Reals.

Notation:
The notation for volt is the capital letter A.

4.4 Semantics

As semantic domain, we will use a mathematical domain that is capable of defining and calculating the in section 4.1 defined operations.

Through the semantic mapping, we can map the following abstract syntax elements with semantic domain elements:

Abstract syntax	**Semantic domain**
DCVoltageSource::voltage	U
ElectricalDiagram::current	I
Resistor::resistance and Resistor::name	R_n
Switch::resistance() and Switch::name	R_n
ElectricalDiagram::equivalentResistance	R_{eq}

Furthermore, when calculating the equivalent resistance over a number of resistors it is necessary to know if the resistors are in serial or parallel. In the abstract syntax of the electrical diagram, this can be checked through the wires. A wire with two ends, where both ends are connected with a different resistor, will indicate that at least these two resistors are serial. For parallel, we need a pair of wires, where each wire will contain the same number of wire ends, with as minimum number of wire ends three. Furthermore, both wires must have the same connected resistors for each wire end, with exception to one wire end.

4.5 Examples

We will now build Electrical Diagram Models based on the Electrical Diagram Metamodel. Therefore, we will implement the example circuits given in Figure 57.

Example 1

The first example we will implement is shown in Figure 60. In the electrical circuit, we are setting the DC voltage source with a voltage value of 10 V. Furthermore, we are setting each resistor with an Ohm value, R_1 with a resistance value of 500 Ω, R_2 and R_3 with a resistance value of 1000 Ω.

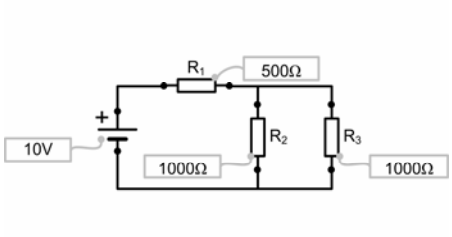


Figure 60 : Example 1 electrical circuit

The representation as shown in the Figure 60 is the one where the abstract syntax is mapped with the concrete syntax. For giving a more precise view of how the abstract syntax is instantiated, we will construct the electrical circuit as object diagram, as shown in Figure 61.

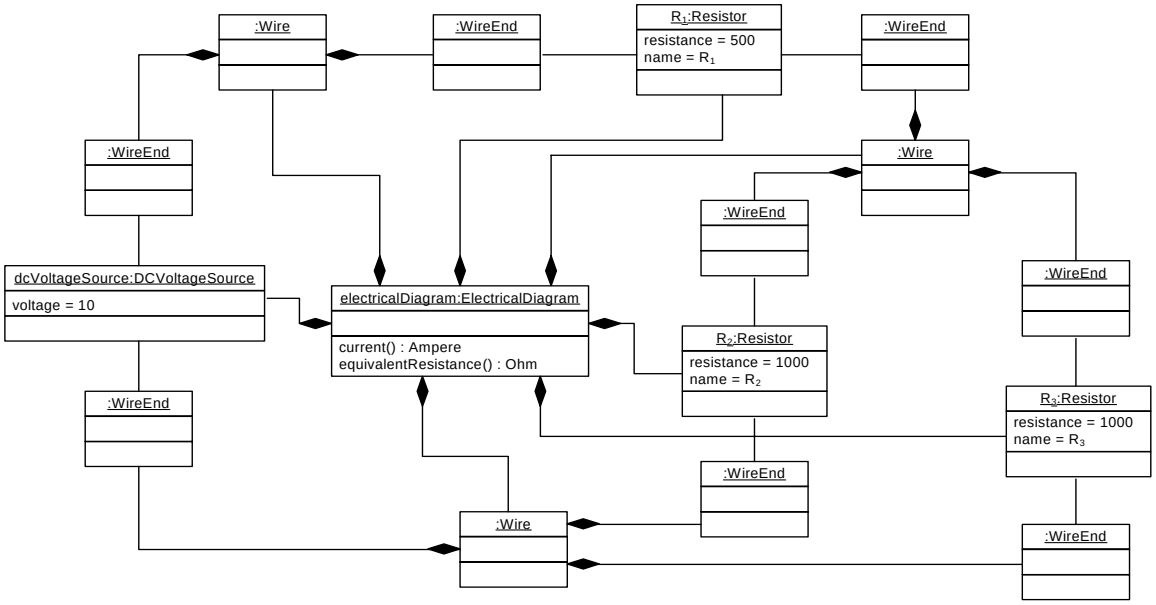


Figure 61 : Electrical circuit as object diagram

After mapping the in Figure 61 defined object diagram with the concrete syntax the model will be representing as in Figure 60. The other mapping that is needed is the semantic mapping, which maps the abstract syntax with the semantic domain. Using the semantic mapping we can represent the abstract syntax as a mathematic formula. In Figure 62, the layer M1 shows both representations of the abstract syntax.

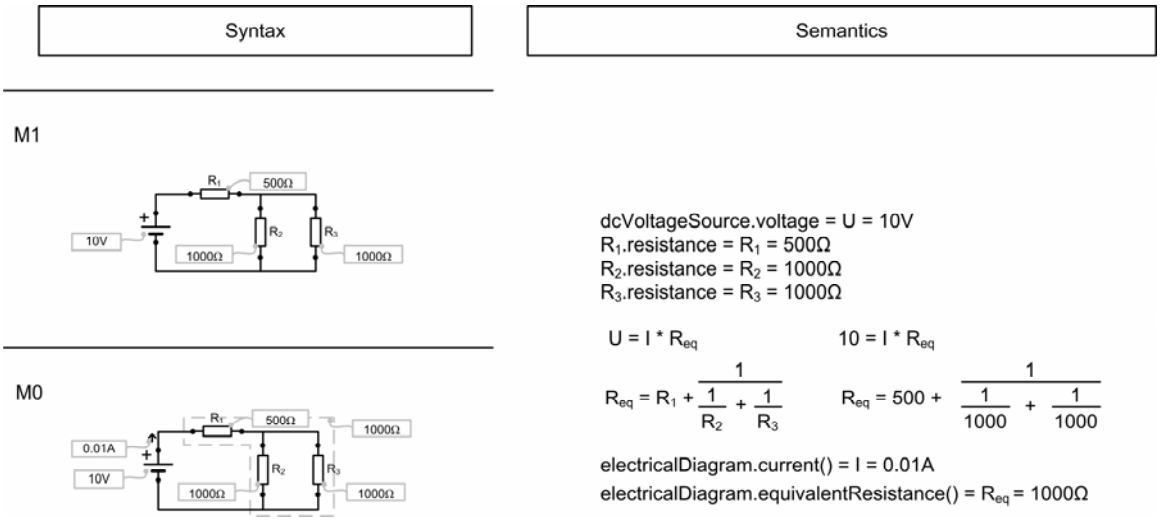


Figure 62 : M1 and M0 layer for electrical circuit example 1

Now the M1 layer is completely defined, the model can be instantiated as runtime. In the runtime we can query the model, which results in getting the values as based on the given values in the model, as shown in Figure 62 representing in the M0 layer. The queries that can be preformed on the model are electricalCircuit.current for the flowing current and electricalCircuit.equivalentResistance that gives the equivalent resistance, according the semantics the flowing current is 0.01 A and the equivalent resistance is 1000 Ω.

The second example will show how we can have multiple instantiations as the runtime. Therefore, the example will have a switch Component that can be changes at runtime. The abstract syntax will be almost the same as the first example, and therefore we will directly continue with the semantic definition for the syntax. The Figure 63 is showing the complete overview of model implementation and runtime view. In the

semantics the resistance operation of the S_1 Switch is now mapped with the R_1 resistance defined in the semantic domain. The defined model can now be instantiated as runtime. The M0-1 layer as representing in the Figure 63 is an instantiation based on the model where the S_1 Switch will have is default value. If we now use the S_1.switch query, the runtime will change as shown in the M0-2 layer.

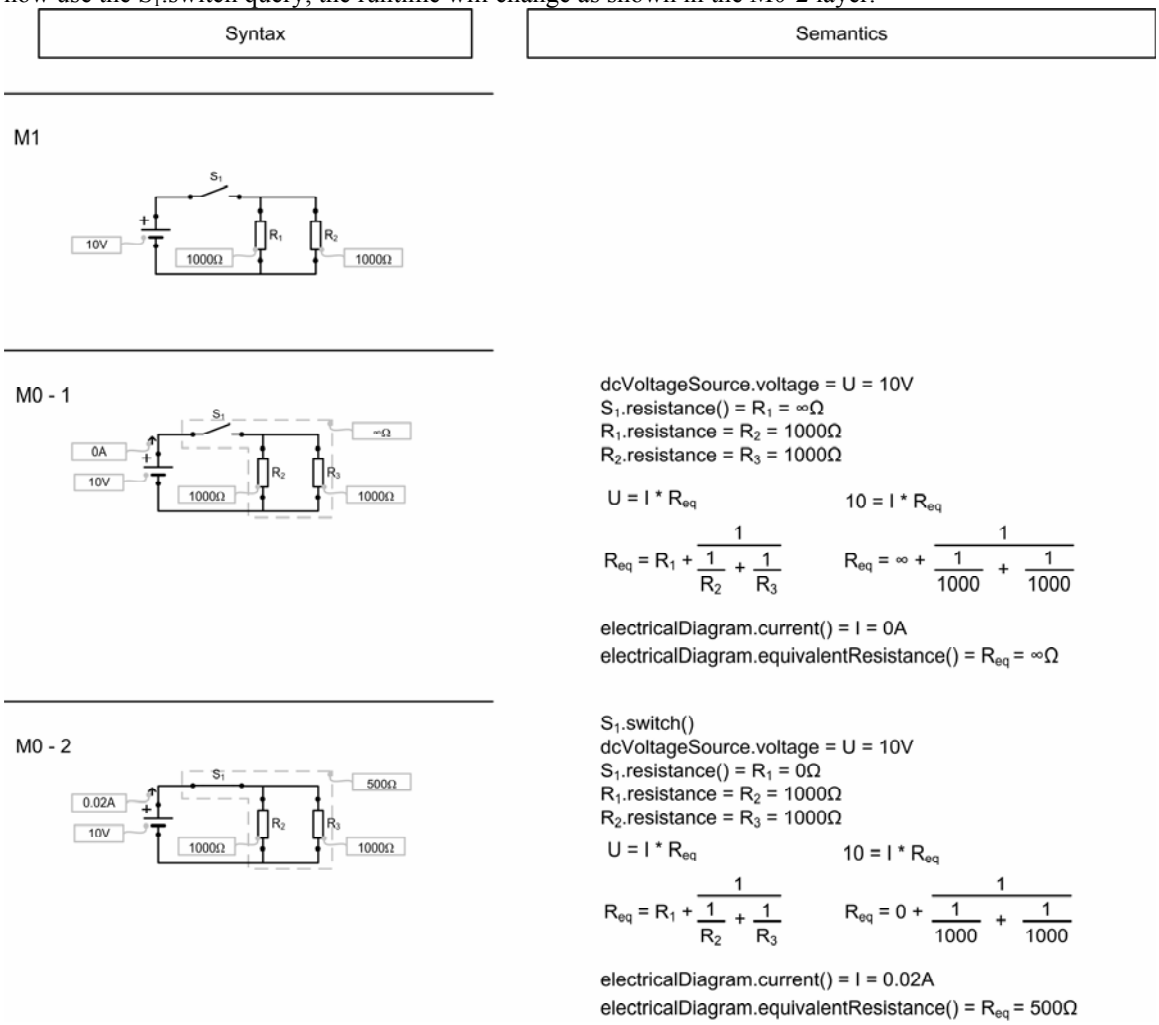


Figure 63 : M1 and M0 layer for electrical circuit example 2

5

Related work

In this section, we will mention related work according to the MOF. As related work we can distinguish the following categories: MOF specification series, compliant tools, MOF metamodels, and related approaches according to the MDA.

5.1 MOF specification series

In this thesis we have described the MOF 2.0 Core, which is part of the MOF 2.0 specification series. The MOF 2.0 specification series contains five specifications in total. The Core specification defines the most essential parts of the MOF 2.0. The other specifications are extensions on the Core specification. These other specifications are the MOF Query/Views/Transformations, the MOF Versioning and Development Lifecycle, the MOF IDL Mapping, and the MOF XMI Mapping. Next, we will briefly discuss each specification.

5.1.1 MOF Query/View/Transformation

Model transformation plays an essential role in the MDA. The MOF Query/View/Transformation (QVT) specification [30] defines a language for describing transformations between models. The defined language has a hybrid declarative/imperative nature, meaning that the specification defines a combination of a declarative and imperative language. The Figure 64 shows the exact architecture of these languages.

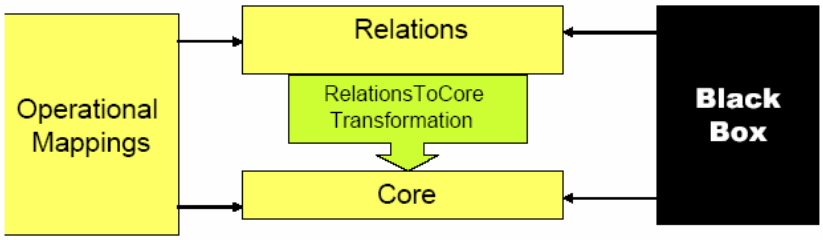


Figure 64 : Relationships between QVT metamodels

The declarative part of the language defines the Relation and Core metamodel. The Core metamodel should not be confused with the Core as defined in the MOF. The Relation metamodel represents a high level language and the Core metamodel represents a low level language. The defined RelationsToCore transformation is capable of translating Relation models into Core models. The imperative part of the language is Operational mappings, an imperative language for defining Relations or Core based metamodels, and Black box, which provides a plug-in facility. The goals for the Black box implementation are [30]:

- It allows complex algorithms to be coded in any programming language with a MOF binding
- It allows the use of domain specific libraries to calculate model property values.
- It allows implementations of some parts of a transformation to be opaque.

The current version of the MOF QVT depends on the EMOF package of the MOF 2.0 Core specification [28] and the OCL as defined in the OCL 2.0 specification [20].

5.1.2 MOF Versioning and Development Lifecycle

The purpose of the MOF Versioning and Development Lifecycle specification is [29]: *"to manage the co-existence of multiple versions of metadata in a MOF and their inclusion in different configurations"*. So this specification is concentrated on the evolution of the system during its development. In the development of the system, changes are made to improve the system, for example the change in lifecycle, as going from a draft system to a live system. On the other hand, we can have small changes to the system in the form of fixing bugs. In most cases it is useful to save each state in developing the system, and make it possible to return to a specific point in the development of the system. Therefore, the MOF series contains this specification.

The current version of the MOF Versioning and Development Lifecycle [29] depends on some packages from the MOF 2.0 Core specification [28].

5.1.3 Mappings

The other two specifications, IDL Mapping and XMI Mapping, are mappings between the MOF specification and other specifications. The definition of mapping according to the OMG [4] is *"Specification of a mechanism for transforming the elements of a model conforming to a particular metamodel into elements of another model that conforms to another (possibly the same) metamodel"*. The mappings defined for the MOF are the XML Metadata Interchange (XMI) and Interface Definition Language (IDL). The difference between both mappings is that the one is for interchange of metadata and the other for defining an interface.

MOF XMI Mapping

Through the XMI specification [32], the MOF is mapped to XML, which provides the ability for defining, interchanging, manipulating and integrating XML data and object. This is used for integrating tools, repositories, applications and data warehouses.

The current version of XMI [32] is 2.1, which is compliant with the MOF 2.0 as defined in the MOF Core specification 2.0 [28].

MOF IDL Mapping

The MOF IDL mapping specification defines the benefit of the mapping as [31]: *"the MOF IDL Language Mapping specification defines a standard mapping from meta-models defined using the MOF Model onto server interfaces. The interfaces themselves are expressed in CORBA IDL that can be generated by instantiating "templates" defined in the MOF specification. The intended semantics of these interfaces are also defined in the specification"*. The MOF IDL mapping is compliant with the MOF 2.0. The mapping rules for mapping MOF elements with IDL elements are defined in OCL 2.0.

5.2 Compliant tools

Tools that are MOF compliant can operate in different areas for providing help in using the MOF specification. Most tools at least support the ability to store MOF compliant metamodels. Furthermore, a tool can be specialized in managing these stored data, by providing an interface that will provide access to the data. Another specialization is providing an environment for creating MOF-based metamodels, like through a graphical interface that provides an environment for drawing the metamodels.

In the section below, we will discuss the MOF compliant tools we found during our investigation. In our investigation, we try to use some of the tools, like MDR and Modx, but the fact that those tools are not compliant with the latest version of the MOF made them not very useful in our research. The tools MOFLON and aMOF2.0forJava are released in the end phase of our investigation, and therefore we were not able to use them.

5.2.1 MDR

The Metadata Repository project (MDR) is a plug-in for the Netbeans tool [16]. It implements the MOF as metadata repository. The tool Netbeans provides a platform, on top of this platform we can write plug-ins, as done with the MDR. The MDR is compatible with Netbeans version 3.6. The MDR implements the MOF 1.4 as defined in the MOF specification 1.4 [9]. The MDR provides a storage mechanism for storing metadata. The metadata can imported into and exported from MDR using XML documents that conforms to the XMI 1.2 standard. The metadata in the MDR can be managed using a Java Metadata Interface (JMI) API.

5.2.2 MOFLON

MOFLON [17] provides a graphical interface for building MOF compliant metamodels. MOFLON is a plugin for the FUJABA tool. The FUJABA tool is a graph transformation tool that implements graph transformation in the form of Story Driven Modeling. The aim of FUJABA is to become UML 2.0 compatible. Therefore, they use the implementation of MOF 2.0 as the first step to become UML 2.0 compatible.

Besides drawing new MOF metamodels with the MOFLON, there is the ability to import MOF metamodel stored as XML documents in the XMI 2.1 format. Furthermore, the drawn MOF metamodels are exportable as JMI code.

5.2.3 ModX

The ModX tool [18] provides a graphical interface for defining two types of models, namely the MOF-based metamodels and models based on the defined MOF-based metamodels. The tool provides also an interface to define a new graphical notation for the metamodel. So besides defining the abstract syntax of the metamodel, we can define the concrete syntax of the metamodel as well.

The metamodels that we can draw, using the ModX tool, are MOF 1.4 compliant. Already existed metamodels can be imported when stored as XML documents in the XMI 1.1 format. Furthermore, the built metamodels can be exported in the XMI 1.1 format.

5.2.4 JMI

The Java Metadata Interface (JMI) provides a mapping between the MOF and Java. According to the [34], *"The JMI specification enables the implementation of a dynamic, platform-independent infrastructure to manage the creation, storage, access, discovery, and exchange of metadata"*. The JMI implementation provides the ability for the generation of a Java interface for each MOF-based metamodel. Furthermore, JMI supports the interchange of metadata and metamodels via XML by using the XMI specification.

Unisys develops JMI in cooperation with Sun Microsystems, Hyperion, IBM, and Oracle. The current version of JMI is compliant with MOF 1.4 and can handle XML documents stored in the XMI 1.1 format.

5.2.5 aMOF2.0forJava

The aMOF2.0forJava provides as the JMI a mapping between the MOF and Java. The difference between both tools is that the aMOF2.0forJava is compliant with the MOF 2.0 instead of the MOF 1.4. The Humboldt University of Berlin develops the tool, with the objective [35]: *"to capture the new modeling capabilities and provide them to the programming environment of a modern implementation language"*. The tool has as main features, as extracted from [35]:

- Implementation of the CMOF model
- Implementation of the UML Infrastructure library. This library is used as a basis for the CMOF model, and it can be used in user meta-models, thus allowing easy development of new languages, based on a given abstract basis.
- XMI import and export of meta-models and models
- Reflection facilities
- Generation of user repositories based on meta-models

- Integration of user code for derived features and operations
- Programming with repositories through an easy to use and type-safe API
- Implemented redefinition and property subsetting semantics

5.3 MOF-based metamodels

The most often used metamodel that is based on the MOF is UML, which is described in section 3.2.3 as part of the MOF Core specification. In this section we will briefly discuss other metamodels that are based on the MOF.

5.3.1 Ontology Definition Metamodel

The Ontology Definition Metamodel (ODM) is a collection of metamodels and mappings between those metamodels that enable ontology modeling. Ontology is a discipline rooted in philosophy, and defined by the ODM specification [11] as:

"An ontology defines the common terms and concepts (meaning) used to describe and represent an area of knowledge. An ontology can range in expressivity from a Taxonomy (knowledge with minimal hierarchy or a parent/child structure), to a Thesaurus (words and synonyms), to a Conceptual Model (with more complex knowledge), to a Logical Theory (with very rich, complex, consistent and meaningful knowledge)."

The ODM uses seven metamodels, which can be grouped into a formal first order and description logics, structural and subsumption / descriptive representations, and traditional conceptual or object-oriented software modeling. Below we describe briefly the seven metamodels as extracted from the ODM specification [11]:

- The group of formal first order and description logics contains the Description Logics (DL) as defined in [39] and the Common Logic (CL) a declarative first-order predicate language as defined by the ISO in [41]. These languages together cover a broad range of representations that lie on a continuum ranging from higher order, modal, probabilistic and intentional representations to very simple taxonomic expression.

- There are three metamodels that represent more structural or descriptive representations, which are the abstract syntax for Resource Description Framework Schema (RDFS) as defined in [42], Web Ontology Language (OWL) [40; 44], and Topic Maps (TM) as defined in [43]. RDFS, OWL and TM are commonly used in the semantic web community for describing vocabularies, ontologies and topics, respectively.

- Two additional metamodels considered essential to the ODM represent more traditional, software engineering approaches to conceptual modeling: UML 2 [45; 6] and Entity Relationship (ER) diagrams. UML and ER methodologies are the two most widely used modeling languages in software engineering today, particularly for conceptual or logical modeling. Interoperability with and use of intellectual capital developed in these languages as a basis for ontology development and further refinement is a key goal of the ODM.

5.3.2 Common Warehouse Metamodel

The Common Warehouse Metamodel (CWM) is a released specification [46] by the OMG for describing metadata interchange between warehouse tools. The OMG describes the CWM as [47]: *"a specification that describes metadata interchange among data warehousing, business intelligence, knowledge management and portal technologies. The OMG MOF bridges the gap between dissimilar meta-models by providing a common basis for meta-models. If two different meta-models are both MOF-conformant, then models based on them can reside in the same repository."* The current version of CWM as defined in CWM specification [46] is compliant with the MOF 1.3.

5.4 Related approaches

All specifications released by the OMG are according to the MDA approach. Besides the MDA approach as introduced by the OMG there exists a variation of approaches in the same area. In this section we will briefly describe some relevant and interesting approaches released for the purpose similar to the MDA.

5.4.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is a related approach for defining a model driven environment, as the MDA provided by the OMG. IBM develops the EMF as plug-in for the open source Eclipse project. The difference between the EMF and the MDA is that the EMF is an already implemented system, where in the case of the MDA the concept exists as an abstract specification. The advantage of the EMF is that it is directly useable as tool.

The following description of EMF is extracted from [36]: EMF is a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Models can be specified using annotated Java, XML documents, or modeling tools like Rational Rose, then imported into EMF. Most important of all, EMF provides the foundation for interoperability with other EMF-based tools and applications.

EMF consists of three fundamental pieces:

- EMF - The core EMF framework includes a metamodel (Ecore) for describing models and runtime support for the models including change notification, persistence support with default XMI serialization, and a very efficient reflective API for manipulating EMF objects generically.
- EMF.Edit - The EMF.Edit framework includes generic reusable classes for building editors for EMF models.
- EMF.Codegen - The EMF code generation facility is capable of generating everything needed to build a complete editor for an EMF model. It includes a GUI from which generation options can be specified, and generators can be invoked. The generation facility leverages the JDT (Java Development Tooling) component of Eclipse.

Three levels of code generation are supported:

- Model - provides Java interfaces and implementation classes for all the classes in the model, plus a factory and package (meta data) implementation class.
- Adapters - generates implementation classes (called ItemProviders) that adapt the model classes for editing and display.
- Editor - produces a properly structured editor that conforms to the recommended style for Eclipse EMF model editors and serves as a starting point from which to start customizing.

The Ecore metamodel used in the EMF is almost similar to the EMOF metamodel as defined in the MOF 2.0 Core. For an exact mapping between the Ecore and the EMOF metamodel we refer to the E-MORF [52].

5.4.2 Linguistic and ontological metamodeling

Künhne and Aktinson [38; 49] introduce some new concepts in their vision on the MDA, mentioned as the Model Driven Development (MDD). These concepts concentrate on a more advanced way of metamodeling, which they distinguish in *linguistic* and *ontological* metamodeling. Linguistic metamodeling is used for defining language type constructs, and ontological metamodeling is used for defining domain specific facts.

Künhne and Aktinson explain the difference between linguistic and ontological with the following example [38]. The distinction of metamodeling in linguistic and ontological is introducing two types of instance-of relationships, namely the linguistic and ontological instance-of. First, we will illustrate metamodeling from a linguistic view, as shown in Figure 65. In Figure 65 a four-layered metamodel hierarchy is illustrated,

where the layers M_2 and M_3 are representing metamodels. The elements in the layers M_1 and M_2 have a linguistic instance-of relationship with the elements in the layer above.

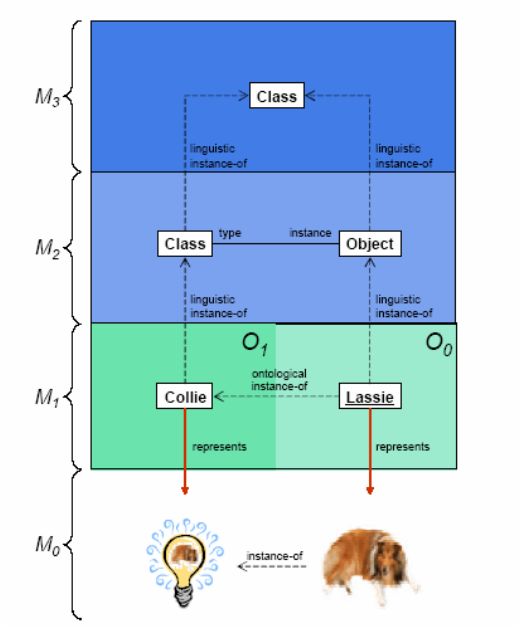


Figure 65 : Linguistic Metamodeling View

Besides the linguistic instance-of relationship, Figure 65 also contains an ontological instance-of relationship, as located in layer M_1 between the elements Collie and Lassie. In the next example we will focus more on the ontological instance-of relationship. Therefore, we will metamodel the same example as shown above, but now from an ontological view.

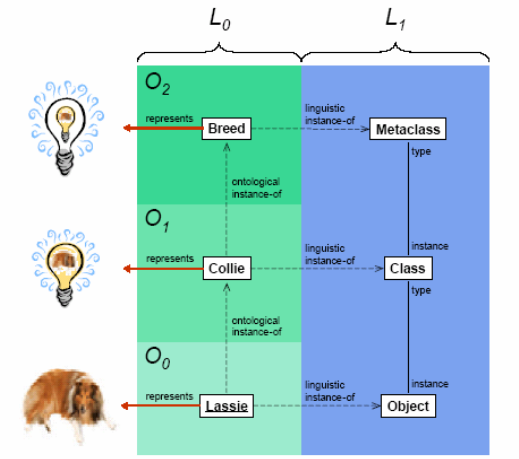


Figure 66 : Ontological Metamodeling View

Figure 66 illustrates an ontological metamodeling view of the example. As shown in Figure 66 the layered structure of the hierarchy is changed. The hierarchy now contains two meta dimensions, namely a linguistic (L) and an ontological (O). Each dimension can be divided into a number of layers. The ontological dimension is divided into three layers, where the elements in the O_1 and O_2 layers are ontological instantiation based on the layer above. The linguistic dimension of the hierarchy is containing two layers, which are vertically positioned in the hierarchy. The three layers of the ontological dimension are part of the L_0 layer. For each element in this layer, a linguistic instance-of relationship can be defined with the meta element in the L_1 layer.

Distinction of metamodeling into linguistic and ontological and using a multiple dimensional hierarchy can provide the ability to define elements in a more natural way. Authors as Bézivin and Lemesle [50] or Geisler et al. [51] also propose the approach of defining more distinctions in metamodeling.

6

Conclusions and recommendations

The MOF as available in the current specification is part of the MDA. The OMG releases the MDA as the new approach for developing software using models and modeling techniques. If we look at the history of software engineering, we can detect that we are continuously searching for a technique that provides a better and more natural approach for defining a system. For example, the introduction of the first FORTRAN compiler in 1957 brought a shift in defining a system in a more abstract way. At this point programmers were able to specify what a machine should do rather than how the machine should do it. (Atkinson and Kühne [38]) After the introduction of the first compiler, the aim to develop a technique that provides the ability to define a system in an even higher abstraction level continues, like with the arrival of procedural and object-oriented languages.

The current techniques used in software engineering, are still concentrating on the implementation of the system. The aim of the MDA is to reach an abstraction level that more concentrates on defining the structure and behavior of the system independent from the underlying implementation technology.

Models and modeling techniques are already often used in all kind of engineering disciplines. In those disciplines models are used for reducing mistakes in working out solutions for a complex problem. The models are the conceptual view of the problem. The aim of modeling is to reach an accurate representation of the problem in the form of models. These models provide a better understanding of the problem, which is necessary before working out the problem as a real world concept. For the notation of the model a representation is needed that easily allows extracting the solution for the problem.

Nowadays, software engineering mainly relies on code-based notations. Code is limited in providing a proper understanding. With modeling we have the ability to define a more sophisticated notation that is closer to our intuition for representing certain concepts.

If we look to the current use of models in the software engineering, we can conclude that in most situations they are only used as documentation. The models are used as the first step in the design of the system. Afterwards, the programmer interprets the models for programming the system. The problem with this method is that the models and the coded system are not connected with each other. This has as result that design changes in the phase of programming have to be manually changed in the models as well. Unfortunately, this is not done in most of the cases, leading to the fact that models easily diverge from code system.

The aim in the MDA is to use the models for automatically generating the system. This will solve the problem that the models are not connected with the system. So the modeled system is so complete that it is an exact representation of the system.

A technique for automatically generating the system from the models, is translating the models into code, which can be used in the traditional way. For example, translate the models into Java code. The Java code can be compiled and runned using the available java compilers and virtual machines. Unfortunately, tools

now available are in most cases only producing code skeletons and fragments, which leads to the fact that additional code is needed for making a complete system. This directly abandons the models from the system. The MDA compliant tools should in the future aim to automaticallygenerate the complete system.

Standard
Standardization is a good approach for codifying techniques, and it helps by specialization of concepts into more sophisticated ones. Furthermore, standardization is useful in realizing interoperability between tools. Important for a standard is that the techniques are correct and understandably defined. The quality of artifacts is stipulated by the quality of the defined techniques in the standard.

The MOF is introduced in 1996 as a request for proposals. In 2006 the latest final specification version is released. In the ten years that the MOF exists the standard has endured some major changes. In the first request for proposal the standard should be part of the OMA, as framework to create, store and manipulate object schemas. Nowadays the MOF is part of the MDA, and is placed as the meta metamodel in the metamodel hierarchy.

All changes in the MOF standard during the past ten years have resulted in the fact that the standard is not stable. Stabilizing a standard is important. It will allow that tools can implement the standard without the consequence that the standard is raptly changed again. Furthermore, it is important for the standard that it is widely accepted, and implemented in many tools. More implementations of the standard will also help in checking of the standard is correct and workable. The first step in stabilizing the MOF standard would be solving the issues as mentioned in appendix D.

OMG's release order of standards is striking. In the case of the MDA introduction, the OMG first released a modeling language, shortly after that they released the language that can be used to define the modeling language. A couple of years later they stabilized a standard that covers the complete modeling vision of which both languages are part. A more logical release order would be to first stabilize the complete vision, after that to release the languages.

Reusing subsets out of other standards is a very often used approach for defining a new standard. The advantage of this approach is that through the reuse standards are easily aligned with each other. The problem with this approach is that the design of a standard heavily relies on the reused architecture. This can lead to the situation where not the best solution is taken but the easiest to use. Furthermore, the result of first releasing, for example, a modeling language and then releasing the language for defining the modeling language is that the design of the latter language heavily relies on the modeling language.

Besides the release order of the OMG, we can also see that the priority of the OMG lies more at standards that already earned their position in the software engineering, like UML. The UML is a self-contained and sophisticated described specification. The MOF specification looks messy in our opinion, especially if we compare it with UML. The UML introduces a specification style that is consistently used in the complete specification. The MOF is not introducing any specification style, but is using one that is similar to the one in UML. Unfortunately, it is inconsistently used in the specification, as mentioned in one of the issues in appendix D.2. It would be more appropriate to release the MOF as self-contained specification. First, the MOF is used for defining UML. Second, the MOF can define other models besides UML. In the current release it is necessary to read a subset of UML to become familiar with the MOF, if the MOF is a self-contained specification this will be avoided.

Metamodel and Metamodeling
The MOF is released as meta metamodel, which means that it is specialized in defining metamodels. With the arrival of modeling in software engineering, the concept of metamodels and metamodeling is introduced as well. The metamodel represents a language that is used for defining models. With metamodeling we define metamodels and will use a meta metamodel as language for defining the metamodels. This is done to provide all types of languages (metamodels) concentrated on different domains, but still based on the same source language (meta metamodel). The idea of defining a domain specific language is that domain experts can better translate their knowledge into a system. Each domain

expert can define a part of the system in the area of his/her specialty. This helps in defining systems that are more sophisticated.

The MOF defines the most essential concepts needed for software engineering and modeling. Therefore, the MOF standard is released in a series of specifications, where each specification is specializing on a concept. The core of the standard is a specification that defines an object-oriented modeling language. Object-oriented languages are nowadays quite often used in the software engineering, and are a good approach for specifying constructions in levels of abstractions. Besides concepts that are already often used in the domain of object-orientation, the MOF defines some concepts that are not very commonly used in this domain, like "subset" and "redefine-of" associations. Especially for these concepts, MOF should provide a good specification that exactly describes what the meaning of the concepts is. The current specification for those concepts is quite limited, which makes it hard to extract correct use of those concepts.

The MOF expects that we talk about one metamodel. Actually, there are two of them, namely an essential one and a more complete metamodel. The idea is that the essential version can describe the more complete metamodel, but in which way and how is not defined in the specification. In some cases this works misleading. It is questionable if metamodels defined based on the essential metamodel are compliant with metamodels defined based on the complete metamodel.

Syntax
The notation as introduced by UML is commonly accepted in the software engineering. For example, class diagrams are very often used in defining the class architecture of a system. The MOF is reusing these diagrams for its own representation and for the representation of the instantiated metamodels. Reusing this notation provides the MOF with a representation that is commonly accepted, and makes the standard and instantiated metamodels easier to understand for software engineers.

Semantics
The semantics in the MOF are weak, and they are not considered the most imported part of the standard, as stated in the UML infrastructure library [6]:"*Currently, the semantics are not considered essential for the development of tools; however, this will probably change in the future.*"

In our opinion a well-defined semantics part of a language is as essential as all other language parts. The best way for defining the semantics is in a formal language that does not contain any ambiguity in its definition. Nevertheless, the best way is most times also the hardest way. Therefore, defining semantics in a formal language is difficult. A more easy way is to define the semantics in a natural language. Especially when still stabilizing the standard it is a good option to provide a relative correct and easy to interpret semantics, but we should carefully consider the limitations of semantics defined in natural language.

With the reuse of the UML infrastructure library, we consider that the semantics of the UML infrastructure library also becomes part of the MOF, although that this is not stated in the MOF standard. The semantics in the UML infrastructure library are relatively well defined and understandable, with some exception, for example, the inheritance of operations is incomplete. If we look to the MOF semantics, we can conclude that these semantics are weakly defined and incomplete. Many constructions are not containing any semantics, and constructions that contain semantics are very weak.

One of the goals of the MOF is to enable interoperability between tools. For realizing this, semantics specified in a natural language are not enough. We will need a more formal definition that provides the ability for checking the compliance and interoperability between tools.

The MOF should improve their semantics. The semantics are essential for obtaining the goals, like enabling interoperability. The first step of improving the semantics will be to write them correctly and completely in precise natural language, as already is done but not so correct and complete. The next step will define them by using a formal language.

Reflection
The reflective ability of UML infrastructure library is defined as [6]:
"A specific characteristic about metamodeling is the ability to define languages as being reflective, i.e., language that can be used to define themselves. The InfrastructureLibrary is an example of this, since it contains all the metaclasses required to define itself. When a language is reflective, there is no need to define another language to specify its semantics. MOF is reflective since it is based on the InfrastructureLibrary, and there is thus no need to have additional meta-layers above MOF."

In other words, the reflective ability of UML infrastructure library is defined as the fact that it contains all the metaclasses required to define itself. Because of this facility, it does not requires another language to specify its semantics.

The definition of the reflective ability defined in the UML infrastructure library contradicts with the definition of semantics and reflection, as given in section 2.3 and 2.8. With the semantics we describe the meaning in terms of the concepts that are already well-defined and well-understood. Therefore, we at least need another language to define the semantics in order to explain them in an already well-defined and well-understood way. This because of it is impossible to explain the meaning of new concepts when only using those concepts.

Furthermore, reflection provides the ability for self-representation of its own behavior and structure. This does not say any thing about the meaning of the system. Therefore, semantics are out of the scope of reflection. So it makes no sense to state that not another language is needed to define the semantics if we have a reflective language.

The metamodel hierarchy
The metamodel hierarchy as introduced by the OMG is criticized a lot in the literature. In most cases the hierarchy is found to be too limited to provide the correct representation. The OMG states relationship between elements in different layers as instanceOf. However, not each layer represents the same concept. It is important to represent concepts as accurately as possible. When relationships are given the same name they should represent exactly the same concepts. In the case of OMGs hierarchy this is doubtful, for example, the instantiation of a metaclass into a class is not the same as the runtime interpretation of a class as an object. In our opinion, a more specialized hierarchy will be useful.

Evaluation
Our first approach for investigating MOFs state of the art is looking for practical usage of the standard. Tools will have a central role in the investigation. The way tools have implemented the standard is the central concern for investigating the practical use of the MOF. Therefore, tools that claim to be MOF compliant are investigated on the degree of supporting the MOF standard. Unfortunately, MOF compliant tools are very rare, and most of the available tools are still very experimental implemented. According to this, we changed our approach for investigating MOFs state of the art. Therefore, we described the practical use in a more theoretical way. With the new approach we were able to describe our interpretation of the standard and build based on this implementation an example MOF-based metamodel. Furthermore, we found some undocumented issues remaining in the standard, which are described in appendix D.

Summary of conclusion
The modeling approach is a good shift in software engineering to obtain a higher level of abstraction for defining a system. With the arrival of the MDA, the modeling concept for software engineering is standardized. The MOF standard is as an object-oriented meta metamodel a good approach for metamodeling. The standard can still use some improvement. The issues as mentioned in the appendix D should be solved. The standard should be more stable, so that more tools are implementing it. The standard should be released as self-contained specification for improving the readability. The standard should improve the semantics. Therefore, the semantics defined in natural language should be complete, and further they should be defined in a formal language. The standard should be more precise about the ability of being a reflective language. The metamodel hierarchy, as used in combination with the MOF, should be revised on its limitations.

7

Glossary

Abstract syntax: defines the structure of the notation.

Axiomatic-top-layer: top layer in a modeling hierarchy described by another language.

Causal connection: connection that will hold if the domain represented by the system and internal structure and behavior of the system is linked in such a way that when one form is changing the other form is changing as well.

Concrete syntax: defines the physical notation.

Interpreter: one who interpret models.

Meaning: an interpreted goal or intent.

Meaningful: having meaning, function, or purpose.

Meaningless: having no meaning or direction or purpose.

Meta metamodel: model representing a language for describing metamodels.

Meta-layer: layer in a modeling hierarchy, will contain metamodels.

Metamodel: model representing a language for describing models.

Modeler: one who models.

Modeling hierarchy: collection of the metamodels, models, and runtime divided into layers.

Modeling: to make a model.

Model-layer: layer in a modeling hierarchy, will contain models.

Purpose: a result or effect that is intended or desired; an intention.

Recursive- top-layer: top layer in a modeling hierarchy described by itself.

Reflective system: defines a representation of its own behavior and structure, which provides the ability to access, reason about, and alter its own interpretation [3].

Runtime-layer: layer in a modeling hierarchy, will contain the runtime view of a model.

Semantic Domain: the well-defined and well-understood concepts, can be used for explaining a new concept.

Semantics: defines the meaning.

Strict modeling hierarchy: modeling hierarchy where each element in a model has at least one relationship with a meta-element.

Syntax: defines the notation.

8

Reference

[1] Bézivin, J., Lemesle, R., Towards a True Reflective Modeling Scheme. In Reflection and Software Engineering, W. Cazzola, R. Stroud and F. Tisato (eds.). Lecture Notes in Computer Science, Vol. 1826, pp.21-28, Springer, Heidelberg, Germany, 2000

[2] Smith, B.C., Procedural Reflection in Programming Languages, PhD Thesis, MIT, Available as MIT Laboratory of Computer Science Technical Report 272, Cambridge, Mass., 1982

[3] Blair, G., Notes on Reflective Middleware. Technical report, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, UK, 1997

[4] Miller, J., Mukerji, J., eds.: MDA Guide Version 1.0.1. Object Management Group, 2003

[5] Gitzel, R., Hildenbrand, T., A Taxonomy of Metamodel Hierarchies, to appear, website: http://bibserv7.bib.uni-mannheim.de/madoc/volltexte/2005/993/, 2005

[6] OMG, UML 2.0 Infrastructure Specification, OMG Adopted Specification, ptc/04-10-14, 2004

[7] Kühne, T., Understanding metamodeling, ACM Press, New YORK, 2005

[8] Atkinson, C., Kühne, T., "Calling a Spade a Spade in the MDA Infrastructure", in: Proceedings of the Metamodeling for MDA First International Workshop, pp. 9-12, York, UK, 2003

[9] OMG, Meta Object Facility (MOF) Specification 1.4, 2002

[10] OMG, Meta Object Facility (MOF) 2.0 Core Specification, ptc/04-10-15, 2004

[11] OMG, Ontology Definition Metamodel, 05-08-01, 2005

[12] Konstantas, D., Interoperation of object oriented applications. In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition, pages 69–95, Prentice-Hall, 1995.

[13] OMG, Meta-Object Facility "Common Facilities RFP-5", cf/96-05-02, 1996

[14] Matula, M., NetBeans Metadata Repository, 2003

[15] Soley, R. M., Stone, C. M., Object Management Architecture Guide, Revision 3.0, 1995

[16] MetaData Repository website: http://mdr.netbeans.org/

[17] MOFLON website: http://www.es.tu-darmstadt.de/moflon/index.html

[18] ModX website: http://noce.univ-lille1.fr/projets/ModX/

[19] Meas, P., Nardi, D., Meta-Level Architectures and Reflection, Elsevier Science Inc., New York, USA, 1988

[20] OMG, OCL 2.0 Specifications, ptc/2005-06-06, 2005

[21] Lemesle, R., Meta-modeling and modularity : Comparison between MOF, CDIF & sNets formalisms, Laboratoire de Recherche en Sciences de Gestion, Université de Nantes, 1997

[22] Harel, D., Rumpe, B., Meaningful Modeling: What's the Semantics of "Semantics"?, IEEE Computer Society, 2004

[23] Rose Model containing UML2 metamodel, InfrastructureLibrary.041004.cat, ptc/04-10-05, 2004

[24] Seidewitz, E., What Models Mean, IEEE Computer Society, 2003

[25] Benyon, D., Information and Data Modelling, second edition, McGraw-Hill, Wokingham, 1997

[26] Stachowiak, H., Allgemeine Modelltheorie. Springer-Verlag, Wien and New York, 1973

[27] Kühne, T., What is a Model?, Darmstadt University of Technology, Darmstadt, Germany, 2005

[28] OMG, Meta Object Facility (MOF) 2.0 Core Specification, formal/06-01-01, 2006

[29] OMG, Meta Object Facility (MOF) 2.0 Versioning and Development Lifecycle Specification, ptc/05-08-01, 2005

[30] OMG, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, ptc/05-11-01, 2005

[31] OMG, Meta Object Facility (MOF) IDL Language Mapping Specification, formal/06-01-02, 2006

[32] OMG, MOF 2.0/XMI Mapping Specification, v2.1, formal/05-09-01, 2005

[33] Gardner, T., Griffin, C., Koehler, J., Hauser, R., A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard, In Workshop on Metamodeling for MDA, pp. 179–197, 2003

[34] Java Metadata Interface (JMI) website: http://java.sun.com/products/jmi

[35] aMOF2.0forJava website:
http://www2.informatik.hu-berlin.de/sam/meta-tools/aMOF2.0forJava/index.html

[36] Eclipse Modeling Framework website: http://www.eclipse.org/emf/

[37] IDL website: http://www.omg.org/gettingstarted/omg_idl.htm

[38] Atkinson, C., Kühne, T., Model-Driven Development: A Metamodeling Foundation, IEEE Software 20(5), pp. 36-41, 2003

[39] Baader, F., Calvanes, D., McGuinness, D.L., Nardi, D., Patel-Scheider, P.F., The Description Logic Handbook: Theory, Implementation and Applications, Cambridge University Press, 2003

[40] Dean, M., Schreiber, G., OWL Web Ontology Language Reference. W3C Recommendation 10, Latest version is available at http://www.w3.org/TR/owl-ref/, 2004

[41] ISO/IEC CD 24707 Information technology – Common Logic (Common Logic) – A Framework for a Family of Logic-Based Languages, Latest version is available at http://cl.tamu.edu/docs/cl/24707-for-CD-Spring-2005.doc, 2005

[42] Brickley, D., Guha, R.V., RDF Vocabulary Description Language 1.0: RDF Schema, Latest version is available at http://www.w3.org/TR/rdf-schema/, 2004

[43] ISO/IEC FCD 13250-2: Topic Maps – Data Model, Latest version is available at http://www.isotopicmaps.org/sam/sam-model/, 2005

[44] Patel-Schneider, P.F., Hayes, P., Horrocks, I., OWL Web Ontology Language Semantics and Abstract Syntax, Latest version is available at http://www.w3.org/TR/owl-semantics/, 2004

[45] OMG, UML 2.0 Superstructure Specification, OMG Adopted Specification, ptc/2004-10-02, 2004

[46] OMG, Common Warehouse Metamodel (CWM) specification, OMG Adopted Specification, ad/01-02-01, 2001

[47] Common Warehouse Metamodel (CWM) website: http://www.omg.org/technology/cwm/

[48] Álvarez, J.M., Evans, A., Sammut, P., Mapping between Levels in the Metamodel Architecture, Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, pp. 34-46, 2001

[49] Atkinson, C., Kühne, T., Rearchitecting the UML Infrastructure, ACM Trans. Modeling and Computer Simulation, vol. 12, no. 4, pp. 290-321, 2002

[50] Bézivin, J., Lemesle, R., Ontology-Based Layered Semantics for Precise OA&D Modeling, Lecture Notes in Computer Science 1357, pp. 151-154, Springer, 1998

[51] Geisler, R., Klar, M., Pons, C., Dimensions and Dichotomy in Metamodeling, 3rd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 1998

[52] Gerber, A., Raymond, K., MOF to EMF: There And Back Again, Proc. Eclipse Technology Exchange Workshop, OOPSLA 2003, Anaheim. USA, pp 66-70, 2003

A. History of released OMG standards

This appendix shows the released OMG standards in a timetable.

	Year																	
Release	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006
OMA		[1]					[2]											
CORBA			[1]															
MDA													[1]		[2]			
UML									[2]				[3]			[5],[6]		
RFP							[1]					[4]						
MOF									[2]			[3]		[4]		[6]		[7]
RFP								[1]					[5]					

Modeling

OMA
[1] OMG Object Management Architecture Guide (OMA Guide), Revision 1.0, 1990
[2] OMG Object Management Architecture Guide (OMA Guide), Revision 3.0, 1995

CORBA
[1] Common Object Request Broker: Architecture and Specification Reversion 1.1, 1991

MDA
[1] MDA Guide Version 1.0, 2001
[2] MDA Guide Version 1.0.1, 2003

UML
[1] Unified Modeling Language (UML) RFP, 1995
[2] Unified Modeling Language (UML) 1.0 and 1.1, 1997
[3] Unified Modeling Language (UML) 1.4, 2001
[4] Unified Modeling Language (UML) 2.0 Superstructure RFP, 2000
[5] Unified Modeling Language (UML) Specification: Infrastructure version 2.0, 2004
[6] Unified Modeling Language (UML) Superstructure version 2.0, 2004

MOF
[1] Meta-Object Facility "Common Facilities RFP-5", 1996
[2] Meta Object Facility (MOF) 1.1, 1997
[3] Meta Object Facility (MOF) Specification version 1.3, 2000
[4] Meta Object Facility (MOF) Specification version 1.4, 2002
[5] Meta Object Facility (MOF) 2.0 Core RFP, 2001
[6] Meta Object Facility (MOF) 2.0 Core Specification, 2004
[7] Meta Object Facility (MOF) 2.0 Core Specification, 2006

B. EMOF

Figure 67 shows the EMOF without the use of any import and merge.

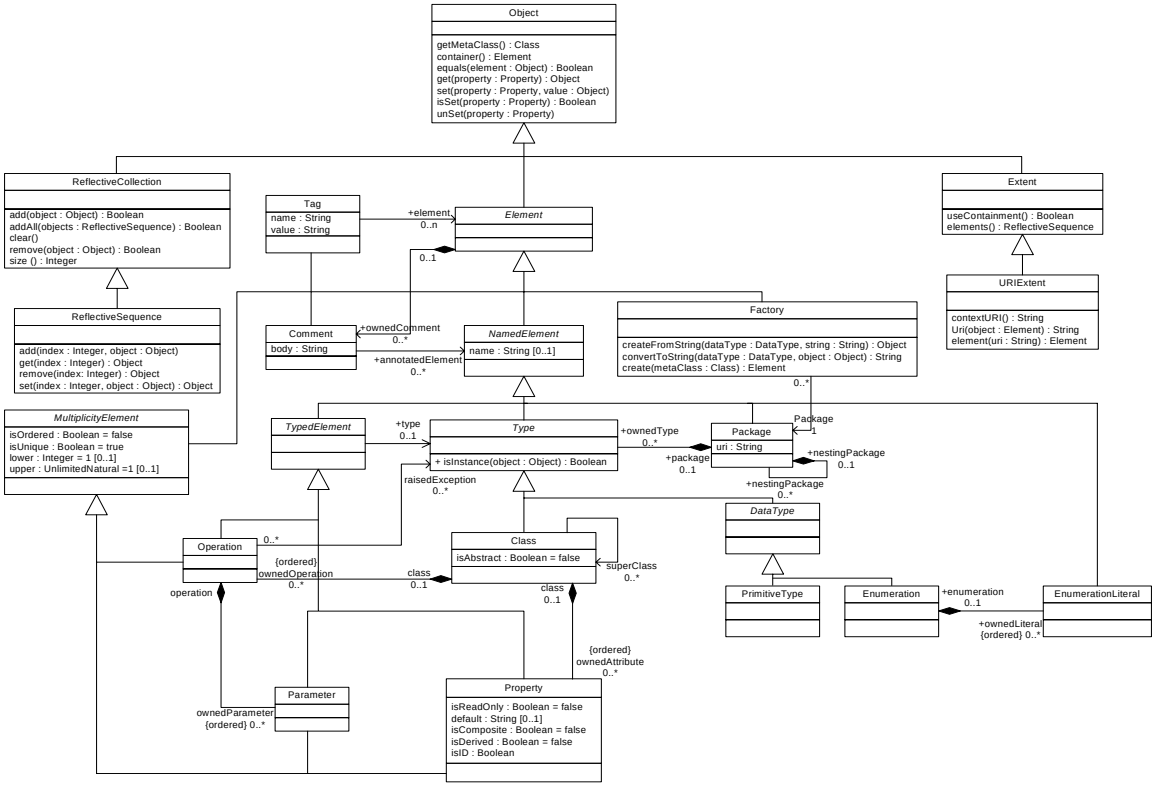


Figure 67 : EMOF without import and merge

C. UML notation for MOF

This appendix contains the explanation for the UML notation that is used as concrete syntax for the MOF.

Class
The notation of a class is a rectangular box with inside the class name. A class can be splits into two types, namely abstract and non-abstract. The representation for defining if a class is abstract or non-abstract is done through the class-name, in case of an abstract class the class-name will be italic and for a non-abstract class the class-name will be normal. Figure 68 will show an abstract and non-abstract class.

Figure 68 : Abstract and non-abstract class

EMOF and CMOF define an abstract or non-abstract class through the class attribute isAbstract. An abstract class will have an isAbstract set to true, and a non-abstract class will have an isAbstract set to false.

Properties, operations, and parameters
For notating properties, operations, and parameters, a textual notation will be used. The textual notation can contain a huge amount of options, and the exact syntax to notate these options will be explained in BNF.

Multiplicity
Each item can have the ability to define a multiplicity, the BNF for multiplicity is:

<multiplicity> ::= *<multiplicity-range>*
<multiplicity-range> ::= *[<lower> '..'] <upper>*
<lower> ::= *<integer>*
<upper> ::= '*' | *<unlimited_natural>*

Where:

BNF	Description [UML]	EMOF	CMOF
<lower>	The lower bound of the multiplicity interval	MultiplicityElement::lower	MultiplicityElement::lower
<upper>	The upper bound of the multiplicity interval	MultiplicityElement::upper	MultiplicityElement::upper
<integer>	Representing the integer value	PrimitiveType::integer	PrimitiveType::integer
<unlimited_natural>	Representing the unlimited natural value	PrimitiveType:: unlimitednatural	PrimitiveType:: unlimitednatural

If the <lower> bound is equal to the upper bound, then an alternate notation is to use the string containing just the <upper> bound. For example, "1" is semantically equivalent to "1..1".
A multiplicity with zero as the <lower> bound and an unspecified <upper> bound may use the alternative notation containing a single asterisk "*" instead of "0..*". [6]

Visibility
Properties and operations can have a visibility, the BNF for visibility is:

<visibility> ::= '+' | '-'

Where:

BNF	Description [UML]	EMOF	CMOF
<visibility>	The visibility of the property	X	<<enumeration>> VisibilityKind with literal values: public, private

Property

Properties are used for attributes and associations, depending on there use they have a particular notation. The BNF for property is:

<property> ::= [<visibility>]['/']<name>[':'<prop-type>] ['['<multiplicity>']']
['='<default>] ['{'<prop-property>[','<prop-property>]'}']*

<prop-modifier> ::= 'readOnly' | 'union' | 'subsets' <property-name> | 'redefines'
<property-name> | 'ordered' | 'unique' | <prop-constraint>

Where:

BNF	Description [UML]	EMOF	CMOF
'/'	signifies that the property is derived	Property::isDerived	Association::isDerived Property::isDerived
<name>	The name of the property	NamedElement::name	NamedElement::name
<prop-type>	the name of the Classifier that is the type of the property	TypedElement::type	TypedElement::type
<multiplicity>	multiplicity of the property. If this is omitted it implies a multiplicity of exactly one.	See Multiplicity	See Multiplicity
<default>	An expression that evaluates of the default value or values of the property.	Property::default	Property::default
readOnly	Means that the property is read only	Property::isReadOnly	Property::isReadOnly
union	Means that the property is a derived union of its subsets	X	Property::isDerivedUnion
subsets <property-name>	Means that the property redefines an inherited property identified by <property-name>	X	Property::subsettedProperty
redefines <property-name>	Means that the property redefines an inherited property identified by <property-name>	X	Property::redefinedProperty
ordered	Means that the property is ordered	MultiplicityElement::isOrdered	MultiplicityElement::isOrdered
unique	Means that there are no duplicates in a multi-valued property	MultiplicityElement::isUnique	MultiplicityElement::isUnique

Examples:

Property as attribute:
aProperty : Property [1..2] {ordered, unique}

Property as association:
aProperty 1..2 {ordered, unique}

Attribute

Properties can be used as attributes; therefore, the property is notated in the class, as shown in Figure 69.

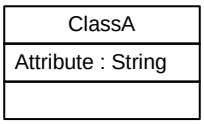


Figure 69 : Attribute

EMOF and CMOF define attributes as a Property with as type an instantiated DataType. A class can own the attributes by means of the association ownedAttribute.

Operation and Parameter

Operations and parameters are always used in combination, where the parameter is a part of the operation. Operations in combination with parameters are notated in the class. The BNF for operation and parameter is:

<operation> ::= [<visibility>] <name> '(' [<parameter-list>] ')' [':' [<return-type>] '{' <oper-property>[',' <oper-property>] '}']*

<parameter-list> ::= <parameter> [','<parameter>]

<parameter> ::= [<direction>] <parameter-name> ':' <type-expression> ['['<multiplicity>']']['=' <default>] ['{' <parm-property> [',' <parm-property>] '}']*

<oper-property> ::= 'redefines' <oper-name> | 'query' | 'ordered' | 'unique' | <oper-constraint>

Where:

BNF	Description [UML]	EMOF	CMOF
<name>	The name of the operation	NamedElement::name	NamedElement::name
<return-type>	The type of the return result parameter if the operation has one defined	TypedElement::type	TypedElement::type
<direction>	'in' | 'out' | 'inout' (defaults to 'in' if omitted)	X	<<enumeration>> ParameterDirectionKind
<parameter-name>	The name of the parameter	NamedElement::name	NamedElement::name
<type-expression>	An expression that specifies the type of the parameter	TypedElement::type	TypedElement::type
<default>	An expression that defines the value specification for the default value of the parameter	Parameter::default	Parameter::default
<parm-property>	Indicates additional property values that apply to the parameter	See Multiplicity	See Multiplicity
redefines <oper-name>	Means that the operation redefines an inherited operation identified by <oper-name>	X	Operation::redefinedOperation
query	Means that the operation does not change the state of the system	X	Operation::isQuery
ordered	Means that the values of the return parameter are ordered	MultiplicityElement::isOrdered	Operation::isOrdered
unique	Means that the values returned by the parameter have no duplicates	MultiplicityElement::isUnique	Operation::isUnique

oper-constraint is missing from above table.
The notation for an operation and parameter is as follow:

operation(object : Object [1..2] {ordered, unique}) : Object [1..2] {ordered, unique}

EMOF and CMOF define the operations and parameters as Operation and Parameter class. A class can own operations and parameters by means of the association ownedOperation.

Associations between classes
Associations are notated as a line between two classes, where each association end can have a property, as shown in Figure 70.

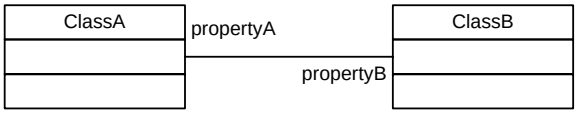


Figure 70 : Association

EMOF defines an association as two properties that are paired with each other. Therefore, the opposite attribute of both properties are set with the opposite property. CMOF defines an association with the class Association, this class contains both properties that form the association. Both properties are defined in the same way as with the EMOF.

We can distinguish the following special associations, which can be used between classes.
Composite
Special association between two classes, where one class is the composed class and the other the composing class, as shown in Figure 71.

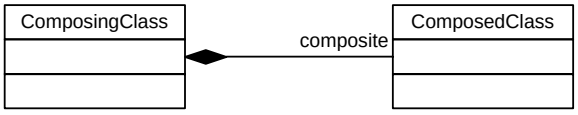


Figure 71 : Composite association

EMOF and CMOF define composite through the property attribute isComposite. Setting the property which owning the composed class with isComposite is true, will result in representing the filled diamond next to the composing class.

Navigate
The navigate association will point to the navigated class, as shown in Figure 72.

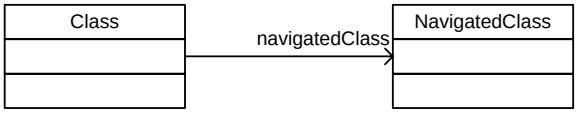


Figure 72 : Navigate association

EMOF defines a navigate association through assigning in the class an attributes with as type an instantiated Class. CMOF defines a navigate association through associate the navigated property with the association by means of the association navigableOwnedEnd.

Inheritance
Inheritance is a special association between two classes, where one class is the inherited class and the other the inheriting class. The representation of inheritance is given in Figure 73.

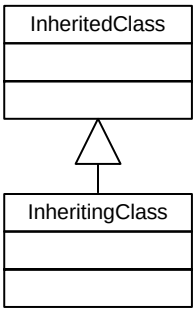


Figure 73 : Inheritance

EMOF and CMOF define inheritance by means of the class association superClass. The superClass association is owned by the inheriting class and will point to the inherit class.

Package

A package is shown as a large rectangle with a small rectangle (a "tab") attached to the left side of the top of the large rectangle. The name of the package is placed within the large rectangle if the members of the package are not shown within the large rectangle. Otherwise, the name of the package is placed within the tab, as shown in Figure 74 [6].

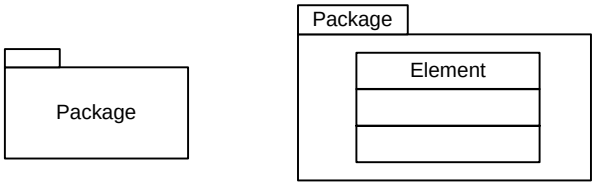


Figure 74 : Package

EMOF and CMOF define this by means of the class Package.

Instance

An instance is using the same notation as Class, with those differences that the name is underlined and will contain name of the instantiated class as well. The instance name and instantiated class name are separated with a colon ':'. The instance contains all attributes and operations direct owned and inherited by the instantiated class. The name of the attributes and operations are enlarged with the originated class name, both names are separated with a point '.'. Figure 75 is showing an instance example.

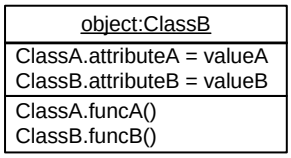


Figure 75 : Object

D. Issues

D.1. Issues UML infrastructure library

The following issues are related to the structure of the Basic package. The issues did arise after comparing the infrastructure specification [6] with the Rose model [23] implementation of the infrastructure.

Issue: Unclear relationship between the Basic and Abstractions packages
1) According to the infrastructure specification [6] the Basic package is using metaclasses from the Abstractions package, as indicated by the following text.
"Basic also contains metaclasses derived from shared metaclasses defined in packages contained in Abstractions. These shared metaclasses are included in Basic by copy."[6 page 91]
First, the mentioned copy construction is not defined in the infrastructure. Second, in contrary to the copy definition, the Rose Model [23] of the infrastructure defines the deriving of metaclasses as import on the package Abstractions::Elements and Abstractions::Multiplicity. (see Figure 76)

2) Furthermore, the infrastructure specification described the reuse of the package Abstractions::Comments as followes.
"Basic::Comment reuses the definition of Comment from Abstractions::Comments." [6 page 92]
The Rose Model [23] does not contain this import.

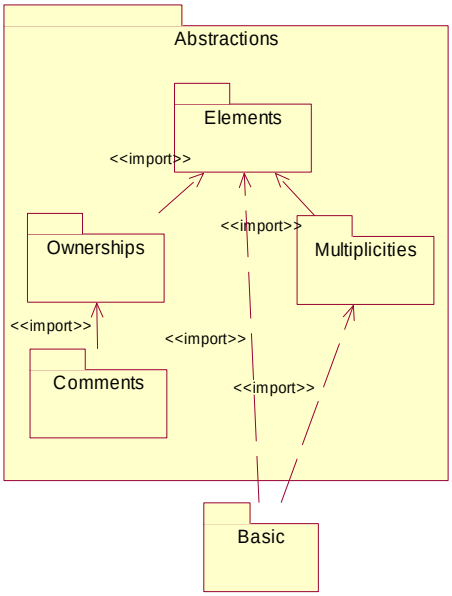


Figure 76

3) The infrastructure specification described the Basic::MultiplicityElement as the reuse of Abstractions::MultiplicityElement:
"Basic::MultiplicityElement reuses the definition from Abstractions::MultiplicityElement"[6 page 97]
The Abstractions package does not contain an Abstractions::MultiplicityElement. Instead of, the Abstractions package does contain an Abstractions::Multiplicities::MultiplicityElement and an Abstractions::MultiplicityExpressions::MultiplicityElement. Owing to the import of Abstractions::Multiplicities the Abstractions::MultiplicityElement should be Abstractions::Multiplicities::MultiplicityElement.

Issue: Description of Element

The infrastructure specification [6] described the metaclass Element as followes:

"Element is an abstract metaclass with no superclass. It is used as the common superclass for all metaclasses in the infrastructure library." [6, page 45 and page 93]

Both packages, Abstraction and Basic, are using the same definition for Element. Therefore, it is logical to assume that both packages will contain their own class Element, as shown in Figure 77.

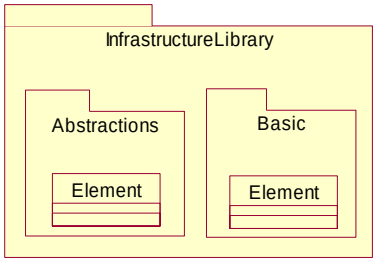


Figure 77

The Rose Model [23] specifies one single class Element, a metaclass that is part of Abstractions. The exact name is Abstractions::Elements::Element. The Basic package imports this metaclass. (see Figure 78). We assume this is the correct interpretation, therefore the text on page 93 should be changed accordingly.

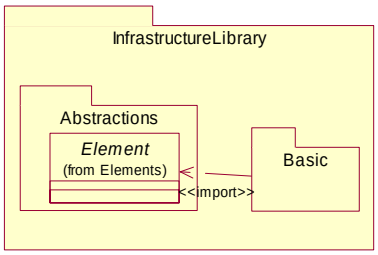


Figure 78

Issue: Element and Comment in Basic

The definition of the classes Element and Comment in the Basic package is ambiguous. The Basic package imports Abstractions::Elements::Element and Abstractions::Comments::Comment. An inheritance relationship and an Association called ownedComment is introduced between Element and Comment in the package Basic. However, these relationships were already defined for these classes in the package Abstractions (see the top two diagrams in Figure 4). Therefore, the complete model of Element and Comment in the Basic package is the model shown in Figure 79, clearly showing a redundant association called ownedComment, and a redundant inheritance relationship between Abstractions::Elements::Element and Comment.

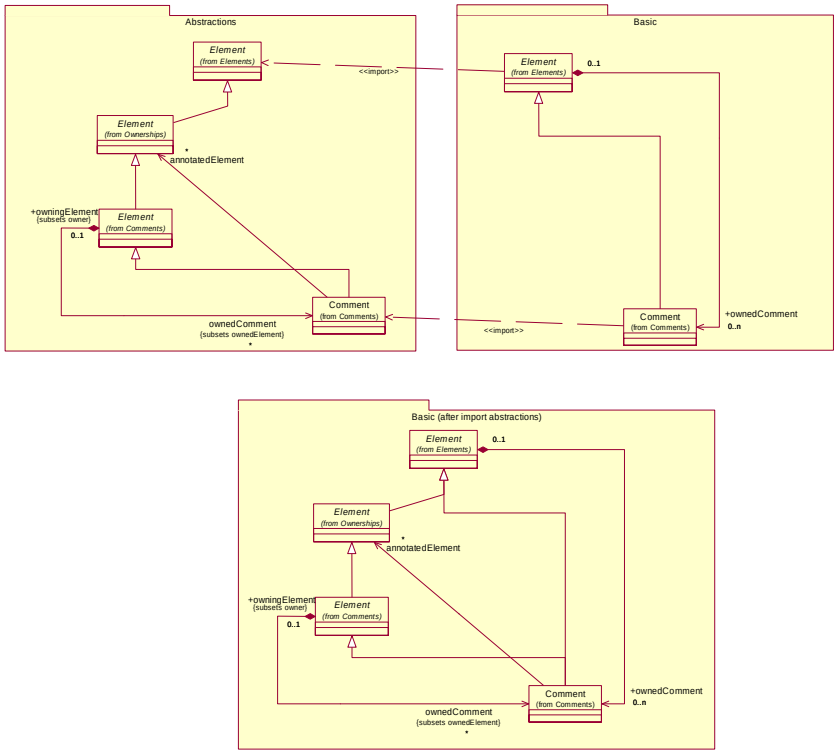


Figure 79

D.2. Issues MOF

The following issues are according the MOF 2.0 specification [10]. For each issue the changes are given according to the MOF 2.0 specification [28].

Typos issues EMOF

Issue – Common::ReflectiveCollection operation remove [10 page 36]:
The syntax for Common::ReflectiveCollection defines the operation remove as:

"remove(object : Object) : Boolean"

The explanation for Common::ReflectiveCollection defines the operation remove as:

"remove(object:Object) : Object
Removes the specified object from the collection. Returns true if the object was removed."

The syntax and the explanation should define the same operation. Therefore, only one of these definitions can be correct.

According [28]: No change.

Issue – Identifiers::Extent operation elements [10 page 32]:
The last sentence of the explanation for the operation elements is:

"See Chapter 4, "Reflection" for a definition of reflectiveSequence"

Chapter 4 of the MOF 2.0 core [10] is about "Terms and Definitions" and does not cover anything about reflectiveSequence. The Chapter 9 is about "Reflection", and Chapter 10 "Identifiers" explains the ReflectiveSequence element in detail, as Chapter 10.5.2 "ReflectiveSequence".

According [28]: No change.

Issue – EMOF Constraints [10 page 45]
The referred chapter in the sixes constraint is incorrect. The referred chapter should be Chapter 15, instead of Chapter 1.

According [28]: Chapter number is changed in 15.

Issue – Class Specification Structure is missing [10 page 35, 36]
The sections "10.5.1 ReflectiveCollection" and "10.5.2 ReflectiveSequence" are not described according the class specification structure.

According [28]: No change.

Issue – Explanation Type is missing. [10 page 25]
In chapter "9 Reflection" the explanation for type is missing.

According [28]: No change.

Design Issues MOF

Issue – addAll operation of ReflectiveCollection [10 page 35]
For using the addAll operation for the ReflectiveCollection class the argument objects should be of the type ReflectiveCollection.

According [28]: No change.

Issue – inheritance operations [10 page 36]
The ReflectiveSequence class does have as superclass the ReflectiveCollection and redefines the operations introduced in ReflectiveCollection, as described as follows:

"Behavior of particular operations defined in ReflectiveCollection is the following when applied to a ReflectiveSequence:

add(object: Object): Boolean
Adds object to the end of the sequence. Returns true if the object was added."

The ability that inherited operations can be redefined in the inheriting class is not defined in the semantics for inheritance.

According [28]: No change.

Issue – Object Capabilities, Object::getType(): Type [10 page 64]
The Reflective signature interpreted/modeled as the equivalent operation on the Instance model does contain the following reflective capability:

"Object::getType(): Type modeled as ObjectInstance::getType(): Type
post: result = self.classifier"

The Object as located in Reflection::Object does not contain an operation getType.

According [28]: No change.

Issue – Convenience/helper OCL operations [10 page 64]
The 15.3 Notes provide the following information about the Convenience/helper OCL operations:

"Convenience/helper OCL operations are used: these are defined in 10.8."

If 10.8 is a number for a section in the MOF specification; there is no section 10.8 that defined the Convenience/helper OCL operations.

According [28]: The part *": these are defined in 10.8"* is deleted.

Issue – Return type operations as defined in the Object Capabilities [10 page 64, 65]
The following operations are defined in the 15.4 Object Capabilities as:

"Object::container(): Object modeled as Instance::container(): ClassInstance"

"Object::get(Property p): Element modeled as ObjectInstance::get(Property p): ElementInstance"

In the Reflection package [10 page 25] the Reflection::Element does contain the operations, as container() and get(). In case of the Reflection::Element operations, the return type is not corresponded. In Reflection::Element the operations are defined as:

"container(): Element"

"get(property: Property): Object"

If the 15.4 Object Capabilities mean the same operations these return values should be aligned with each other.

According [28]: No change.